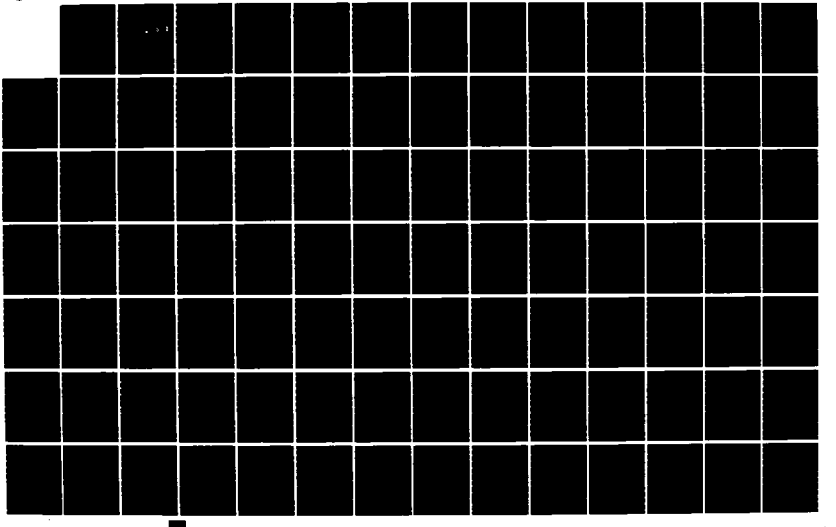
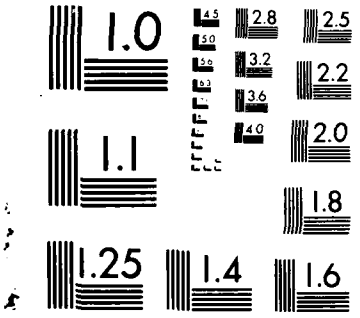
AD-A173 943 EVALUATION OF THE TOOLPACK FORTRAN PROGRAMMING ENVIRONMENT(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA J S KIM JUN 86 1/2

UNCLASSIFIED F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(2)

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

EVALUATION OF THE TOOLPACK
FORTRAN PROGRAMMING ENVIRONMENT

by

Kim, Jung Sik

June 1986

Thesis Advisor: Gordon H. Bradley

Approved for public release; distribution is unlimited.

SECURITY CLASSIFICATION OF THIS PAGE

AD-A173 193

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
2b DECLASSIFICATION / DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b OFFICE SYMBOL (If applicable) 52	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000	7b ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000
8a NAME OF FUNDING / SPONSORING ORGANIZATION 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification) UNCLASSIFIED
Evaluation of the TOOLPACK Fortran Programming Environment

12 PERSONAL AUTHOR(S)
Kim, Jung Sik

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Masters Thesis	FROM _____ TO _____	1986 June 20	

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	TOOLPACK, Programming Environment, Software Library

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

TOOLPACK is a programming environment for the development of medium-size Fortran programs by scientists, engineers and mathematicians. TOOLPACK was developed by a confederation of computer scientists at several government labs and universities in the United States and Great Britain; it was first released in 1985. This thesis is an evaluation of TOOLPACK. It includes a discussion of the installation on the VAX/VMS, benchmarks of tool performance, and a comparison of the users' needs, TOOLPACK goals and TOOLPACK capabilities.

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION	
☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	UNCLASSIFIED	
22a NAME OF RESPONSIBLE INDIVIDUAL Prof. Gordon H. Brdley	22b TELEPHONE (Include Area Code) 408-646-2359	22c OFFICE SYMBOL 52Bz

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete — SECURITY CLASSIFICATION OF THIS PAGE

Approved for public release; distribution is unlimited.

Evaluation of the TOOLPACK Fortran Programming Environment

by

Kim, Jung Sik
Lieutenant, Republic of Korea Navy
B.S., Korea Naval Academy, Jin Hae, Korea, 1979
B.S., Inha University, Inchon, Korea, 1983

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1986

Author: Kim, Jung Sik
Kim, Jung Sik

Approved by:
Gordon H. Bradley, Thesis Advisor

Paul Callahan, Second Reader

Vincent Y. Lum, Chairman,
Department of Computer Science

Kneale T. Marshall,
Dean of Information and Policy Sciences

ABSTRACT

TOOLPACK is a programming environment for the development of medium-size Fortran programs by scientists, engineers and mathematicians. TOOLPACK was developed by a confederation of computer scientists at several government labs and universities in the United States and Great Britain; it was first released in 1985. This thesis is an evaluation of TOOLPACK. It includes a discussion of the installation on the VAX/VMS, benchmarks of tool performance, and a comparison of the users' needs, TOOLPACK goals and TOOLPACK capabilities.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

The author would like to express my thanks for the support and guidance given by my thesis advisor, Professor Gordon H. Bradley, and my second reader LCDR Paul Callahan, in completing this thesis.

The author would also like to thank the Chairman of Computer Science Department, Professor Vincent Lum, for the use of a lot of memory space in the VAX/VMS system; and to thank Andrea Mc Donald, systems programmer of the VAX/VMS, for her expertise and helpful suggestions in the use of the VAX/VMS system.

Additionally, the author would like to thank Dr. Wayne R. Cowell, Argonne National Laboratory, for his assistance and provision of personal research papers.

Also, the author would like to acknowledge the assistance of Mr. Larry Reed, of the National Energy Software Center, for his support during the installation of TOOLPACK.

A very special thanks to my wife, Mee Jeong Kim, and my son, Hak Min Kim, for their patience during these two years.

I. INTRODUCTION

Computer programs have become an indispensable part of research, development, and practice in virtually every area of science, mathematics, and engineering. The development of software in these areas has all the problems with the economic production of correct and reliable software systems that other areas of computer applications have and some special problems such as numerical precision, and the need for significant computation, as well as some unique assets, such as programmers and users with a solid understanding of mathematics. Below we will argue that the scope of computations in this area together with a shared set of problems makes the production of software for science, mathematics, and engineering a prime candidate for the development of software tools to lessen the problems and increase the productivity of programmers working this area.

A. THE IMPORTANCE OF FORTRAN IN SCIENCE, ENGINEERING AND MATHEMATICAL FIELDS

Wayne R. Cowell and Webb C. Miller emphasized the importance of Fortran in the science ,engineering and mathematical fields as follows [Ref. 1: p. 3]:

> Numerical computation is a larger proportion of the total computing activity than is commonly believed. To illustrate, the Department of Defense, for which much computation is of a numerical nature, operates more than twenty five times as many computers as the Internal Revenue Service and the Social Security Administration combined. A study of the matter led John Rice to conclude that numerical computation accounts for about 50 % of the computing expenditures in the United States.

Fortran has been the language of choice for scientists, mathematicians, and engineers for many years. Even through the development of other languages (for example : Prolog, Pascal, and C-language), its position seems solid. For

example, a survey [Ref. 2: pp. 147-162] of commercially available project control systems which are widely used in the construction industry, gave the following data:

- Fortran 21 (41.1 %)
- Cobol 12 (23.5 %)
- Assembler 9 (17.6 %)
- others 9 (17.6 %) ; several software systems were composed of more than one language.

Finally, Fortran is viewed by many as an excellent language for science, mathematics, and engineering computation. The primary reasons for this view are: the wide availability of language processors accepting programs written to the current standard (particularly for advanced scientific machines); the stability of the Fortran implementations; the general purpose nature of many of the facilities offered by the language; the highly efficient object code, particularly for scientific computation; the wide availability of high quality programs and subroutine libraries written in Fortran; and finally, the prevalence of Fortran in scientific computing environments.

B. PROGRAMMING ENVIRONMENTS

It has been estimated that the cost of producing software in the United States in 1980 was between $30 billion and $40 billion [Ref. 3: p. 17]. Other estimates place the cost of producing software at between $40 and $200 per line, depending upon the size and nature of the problem and the software development team. It seems clear that a major reason for this high cost is that nearly all of the activities associated with software production are manual. It has also been clear for well over a decade that many of the manual processes could and should be assisted or replaced by computer supported capabilities. Such computer assistance invariably has taken the form of software designed to help humans perform the activities which are necessary in order

for them to accomplish their software related jobs: coding, testing, documention, transporting, designing and maintenance.

The software programs created to assist in software activities have come to be called software tools. There has been a great surge of interest in creating software tools in the past ten to fifteen years. Unfortunately there has not been the expected decrease in the cost of producing software nor the expected increase in the quality of software products.

In fact, it appears that, despite the apparent appropriateness of using software to help software people, few software tools have been adopted enthusiastically or widely.

The term "programming environment" or "computing environment" is often used denote the set of computer-based file structure, system services (including languages), and system access methods available to assist the programmer [Ref. 4: p. 39].

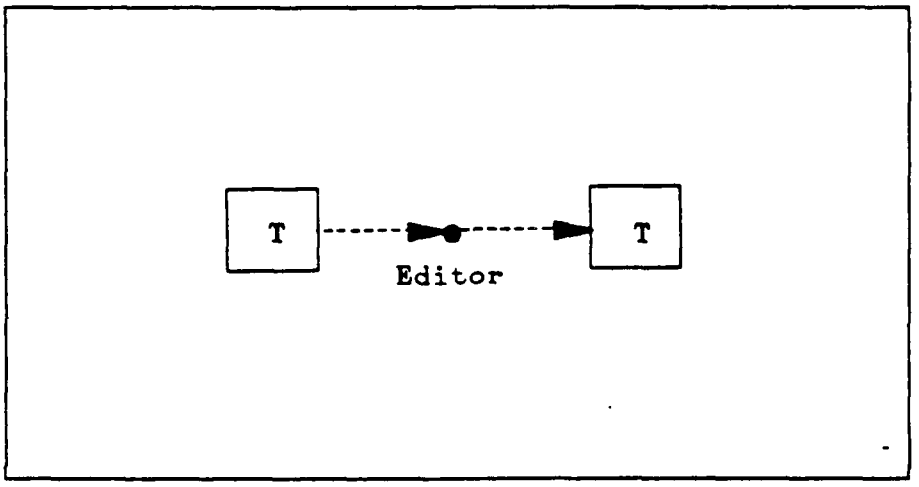
Recently there has been a rapidly growing interest in Programming Support Environments (PSEs). While relatively few such environments actually exist at the present time, it would appear that over the next few years a number of such systems will evolve. Although the development of PSEs is in its infancy it is already the case that it is often very difficult to compare two proposed or extant systems. One problem which contributes to this difficulty is the lack of agreed upon terminology. Another problem is the extreme diversity of functionality which might be desired of or provided by some PSE.

In the following section, there is a discussion of the various kinds of programming activities which a programming support environment might support.

1. Some Programming Activities

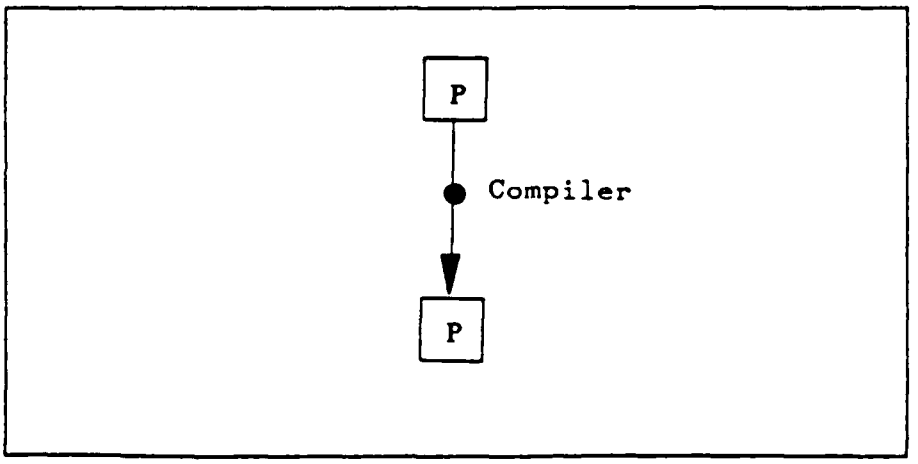
As a preliminary to proposing some terminology and some comparative axes we will present some paradigms of the kind of activities which might be carried out during the life cycle of some engineering projects. We will rely heavily on pictorial representation of the activities in this section.

1) Revision



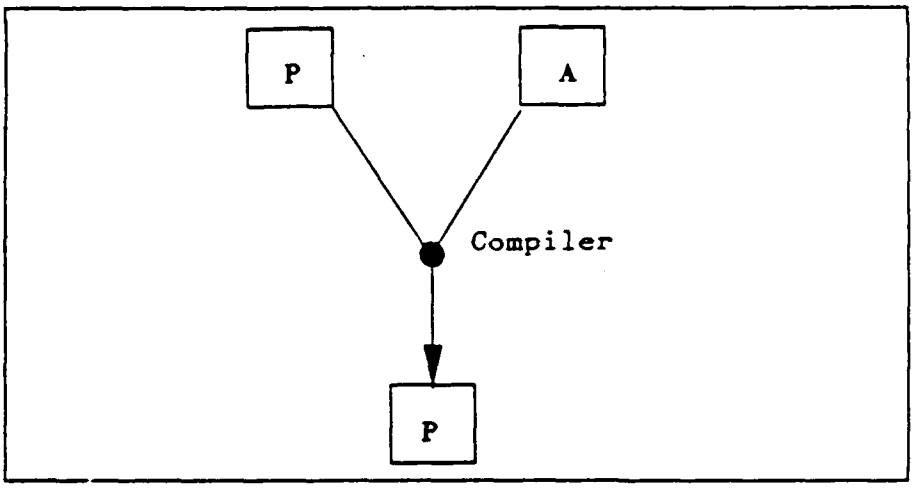
Here the left and right boxes represent text which is modified by using some editor to, for example, make it more complete or to correct certain errors which have been discovered.

2) Compilation



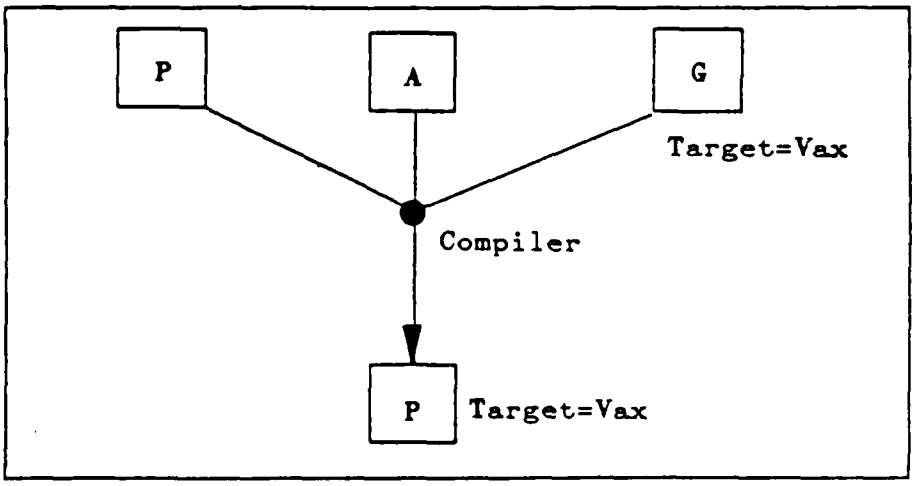
Here the box represents source text for some collection P of program entities and the bottom represents object code.

Another picture of compilation is



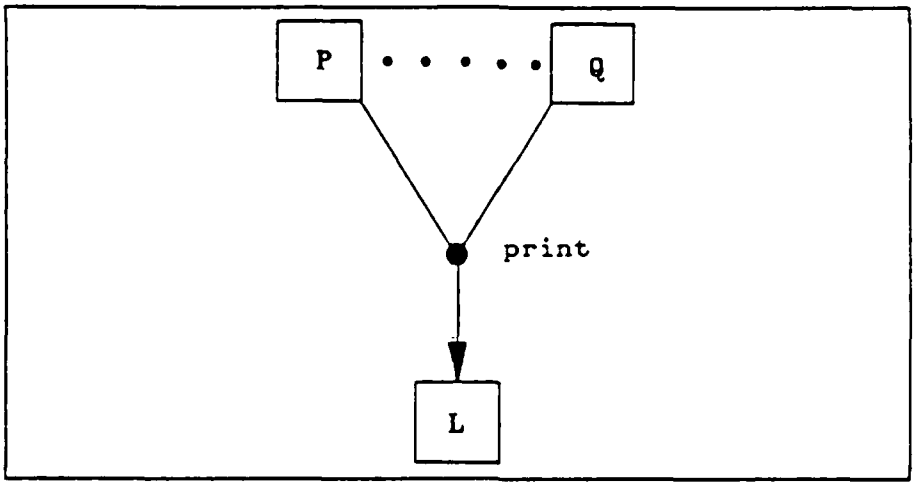
Here the box labelled A represents the results derived by analysis of the program entities (for example, to determine types, identify common subexpressions, assess relative frequency of paths, or what have you).

Yet another picture might be :



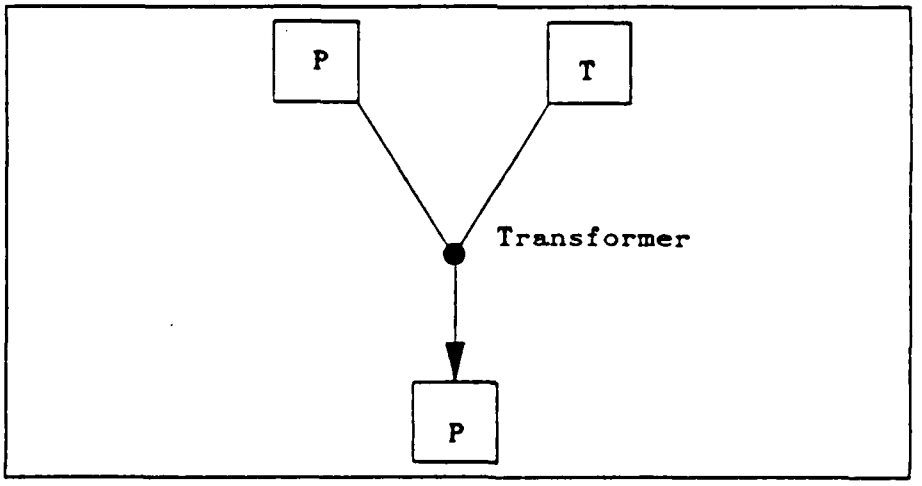
Where the box labelled G represents inputs to the compiler that directs it to produce code for the VAX.

3) Print



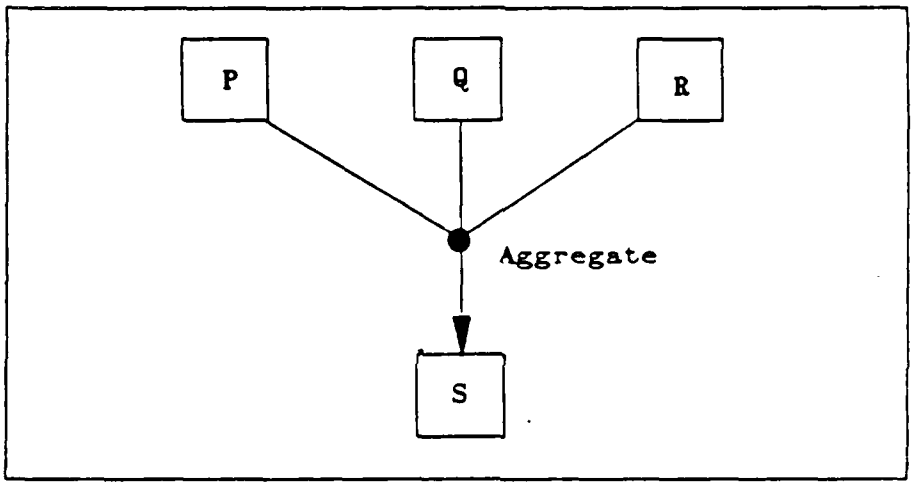
Here the box labelled L represents a (virtual) listing appropriate for the combination of the boxes labelled P,. . ., Q.

4) Transform

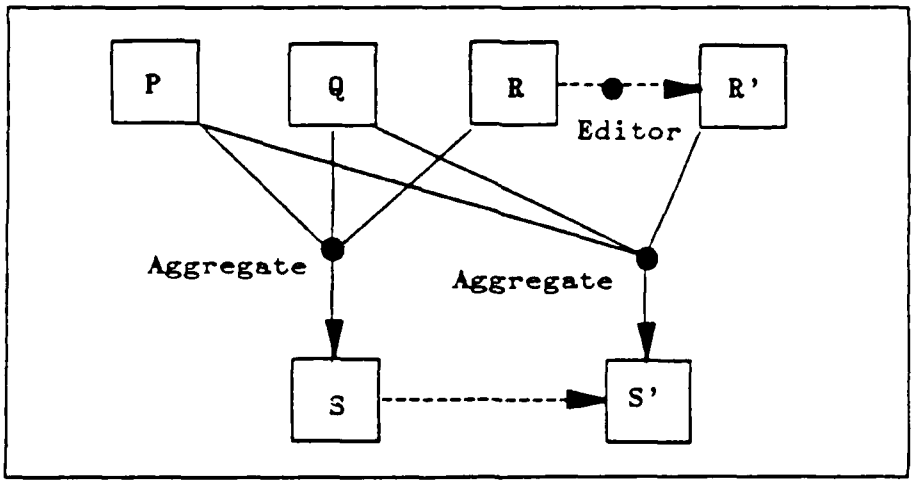


Here we depict the refinement of some collection of program entities (the upper box labelled P) by some transformation (T). Examples include using preprocessors to translate from a problem specific language into some standard high level language, derivation of a procedural specification from a non-procedural specification, using program transformation to refine abstract constructs in a wide spectrum language, and so on.

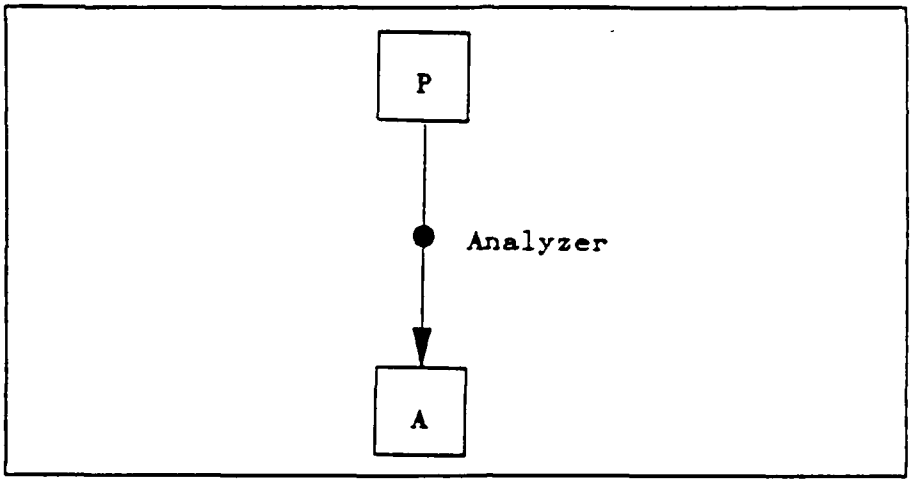
5) Aggregate



Here we might be putting together some "system" (S) composed of components P, Q, and R.
At some later time, after revision of R, we might have:

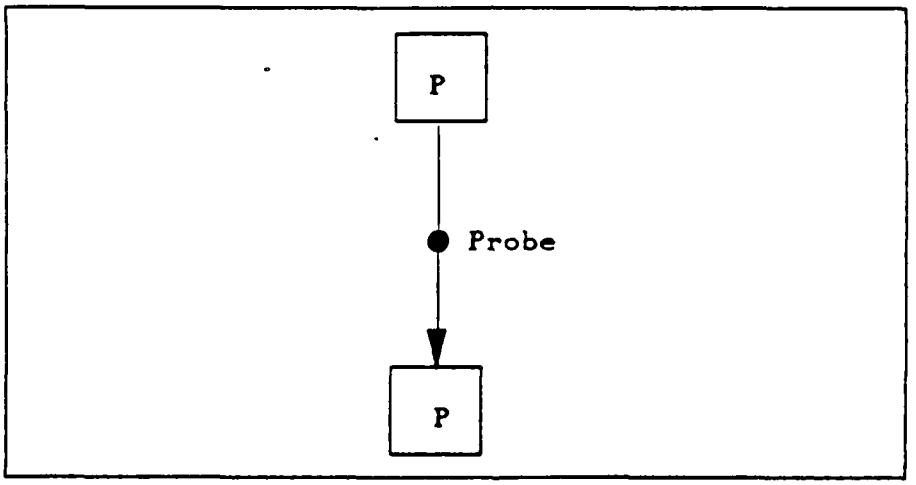


6) Analyze



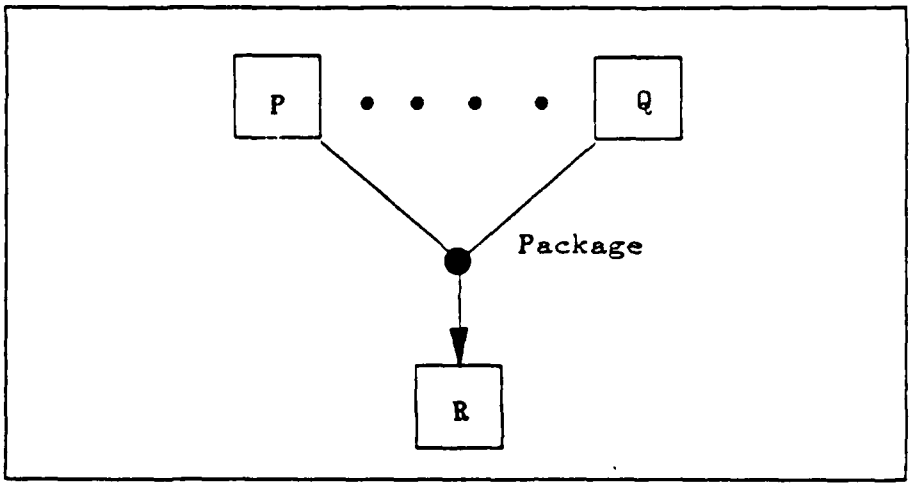
Here the upper box represents some collection of program entities and the lower represents the result of analysis of these entities.

7) Probe



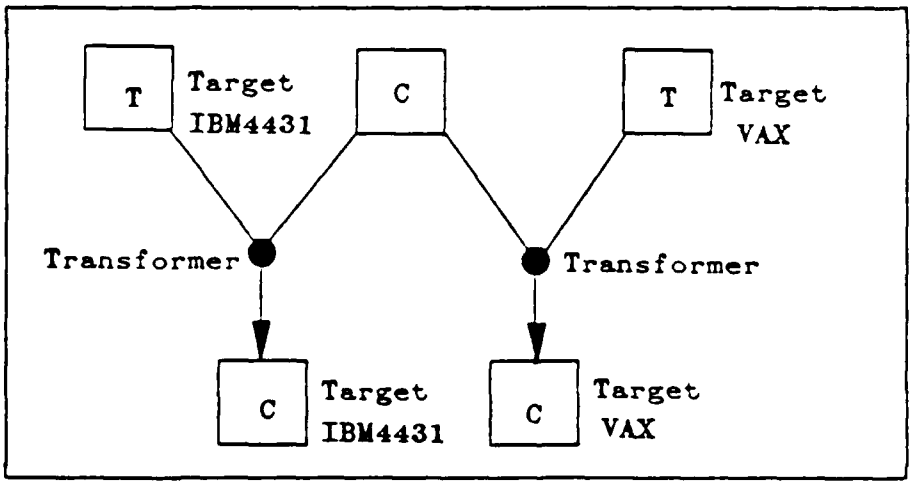
Here the upper box represents some collection of program entities and the lower represents the results of probing these entities during program execution to gather various run-time statistics.

8) Package



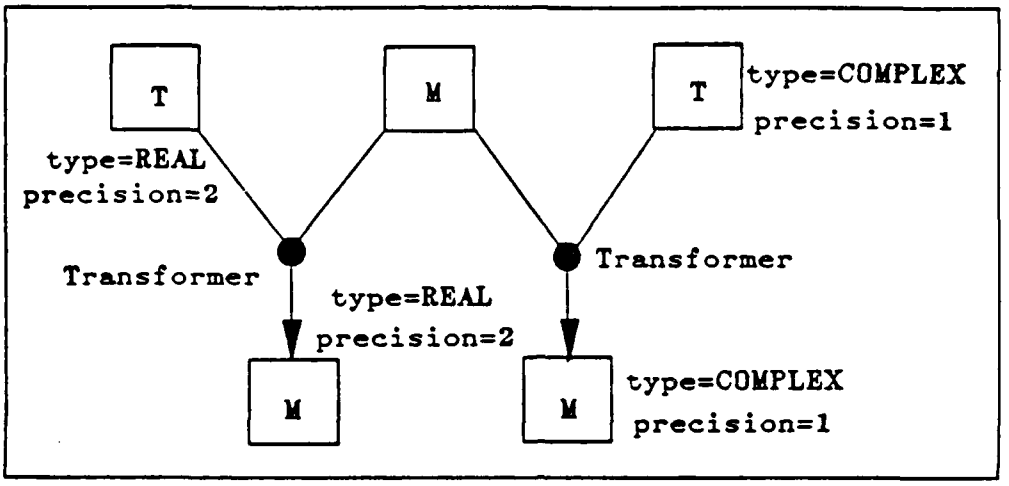
Here we depict the "packaging" of several components (P,. . .,Q) into some derived result (R). Examples include linking object code preparatory to loading, preparing a compiling context, and so on.

9) Families



Here we depict the specialization of a general compiler specification (the upper box labelled C) to two particular compilers (the lower boxes) for, say, an IBM 4431 and a DEC VAX.

Another example is provided by the following:



Here we depict specialization of some general mathematical software package (the upper box labelled M) to two particular packages for specific number types and precisions. We shall speak of the two lower (C) boxes in the preceding and two lower (M) boxes in the above as two members of a family.

2. Some Functional Aspects of Programming Support Environments

Having armed ourselves with a collection of terms to describe various aspects of programming support environments and the activities supported by a PSE we now want to identify a number of axes along which we might measure and compare environments.

1) Language Support

A first question we might ask about a programming support environment is how many programming languages are supported and how well integrated is the support if there is more than one language available.

2) Target Configuration Support

It is becoming widely recognized that the computing complex appropriate for supporting the activities of program development and maintenance must be reasonably large-quite often larger than that appropriate for the operational system being developed. Thus a PSE may support more than one target configuration and, for example, use cross compilers to support target configurations quite different from the host.

3) User Interface

Having a user interface which makes a PSE easy and natural to use is very important.

4) Command Language

By the command language for a programming support environment we mean the set of expressions which the user employs to direct the activities of the PSE. This may be a special language similar to a job control language or may be an extension of a programming language available in the PSE. In general, it is important that the command language be easy to learn and use and that its interpreter be robust.

5) Integration of the Tools

The tools of a PSE are always "integrated" in the sense that they are part of the PSE and they take their inputs from and deliver their results to the software tools. Indeed, the set of programs which can be executed on any operating system are integrated in this very loose sense. Given that a PSE is more structured, it is possible that the tools within a programming environment can be quite highly integrated, resulting in a number of advantages.

6) Granularity of Tools

Typically the tools for program development provided within the context of most operating systems are few in number and large grained. For example, a "compiler" is typically a single tool which operates in several phases: parsing, analysis, optimization, code generation, assembly, and so on. Similarly, an editor typically operates on a complete "file" and leaves no explicit record of what was changed and what remained invariant.

7) Relationships Supported

Conventional programming languages and program development tools provide few facilities for explicitly describing the relationships among the various program entities comprising some software system. Rather, such relationships are usually represented only implicitly.

8) Protection

There are a number of levels at which PSEs may offer facilities which enable modules or parts thereof to be protected from alteration or use by unauthorized parties.

9) Documentation Support

There is a great variety of documentation which is associated with a program system during its lifecycle. Included are items as diverse as requirements documentation, software trouble reports, user manuals, progress reports, time and cost estimates, queries about status, and new release updates. There are, in turn, diverse ways in which a programming environment can support the preparation of and dissemination of documentation.

II. OVERVIEW OF TOOLPACK

Since about 1970 a number of individuals and groups developing mathematical software have begun developing at least moderately elaborate software tools to assist in making their software available on a variety of computers.

As mentioned in the previous chapter, Fortran has been the language of choice for numerical computation, its position seems more solid now than it was ten years ago.

The TOOLPACK project was initiated few years ago with the goal of addressing the problem of inadequate and ineffective use of software tools to develop scientific, engineering, and mathematical software.

The project currently is managed by a confederation of researchers from seven different institutions (Argonne National Laboratory, Bell Telephone Laboratory, Jet Propulsion Laboratory, Numerical Algorithms Group,LTD., Purdue University, University of Arizona, and University of Colorado at Boulder) [Ref. 4: p. 15].

> This confederation has concluded that the problems of ineffective and inadequate utilization of software tools is attributable to the generally poor quality of such tools as well as to the absence of a unifying framework within which they can be evaluated,coordinated compared, and upgraded. Thus the TOOLPACK project has, essentially since its inception, been directed towards the goal of producing high quality tools, and imbedding them in an effective integration framework. Further, the avowed aim of the project has been to make this set of tools and integration framework generally available to the mathematical and scientific software community.

TOOLPACK/1 is the first release of the TOOLPACK Fortran software tools suite. It is the result of an international collaborative project started in 1979. The project was supported by the Department of Energy and the National Science Foundation in the USA and by the Science and

Engineering Research Council in the United Kingdom. In making TOOLPACK/1 available, NAG (Numerical Algorithms Group) is acting as a distribution agent on behalf of the TOOLPACK council.

III. METHOD OF EVALUATION

A. INSTALLING TOOLPACK

An important issue in the use of programs, tools, and environments is how easy it is to install and use them. Installing a system, whether a new one or an existing one that has been modified, consists of the three primary activities of training, conversion, and post-installation review.

Training involves both system operators and users who will use the new system either by providing data, receiving information or actually operating the system. Training the system operators includes not only how to use the system, but also how to diagnose malfunctions and what steps to take when they occur. The users need to be trained to operate the system.

The conversion plan describes all the activities that must occur to install the new system and put it into operation. It identifies the tasks and assigns the responsibilities for carrying them out. The conversion plan should also anticipate the most common problems, such as missing documents, incorrect data formats, lost data, and unanticipated system requirements, and provide ways for dealing with them when they occur.

The post-installation review not only assesses how well the current system is designed and implemented, but also is a valuable source of information that can be applied to the next system project.

We will evaluate the difficulty of installing and maintaining the TOOLPACK environment in chapter 7 (Installation on VAX/VMS).

B. RUN-TIME COMPARISON

This thesis uses the benchmark technique to compare the elapsed CPU times of executing several of the TOOLPACK tools. The selected program sizes are classified into 85 lines of code (LOC), 700 LOC, and 1500 LOC. The overview of the benchmark tests and actual data analysis are described in chapter 8 (Capabilities of System and Tools).

Most operating systems and some special program tools are implemented in their own system level languages or in lower level languages (e.g., Assembler language) to increase the performance. But in order to be portable over several different machines, the TOOLPACK project uses high level languages (Fortran-77 and some Pascal); the use of high level languages causes lower performance.

C. COMPARISON OF THREE ASPECTS

The TOOLPACK project was originally designed for scientists, mathematicians,and engineers not in the computer science field. This thesis will compare the goals of TOOLPACK with the needs of programmers of mathematical software and compare the goals of the TOOLPACK project with the actual capabilities of the TOOLPACK project (See Figure 3.1). To compare the three aspects, some knowledge of them is required. Therefore, the author will introduce each concept in the following chapters.

Chapter 4 will describe user's view (views of scientists, mathematicians and engineers). Chapter 5 will describe the initial goals of the TOOLPACK project. Chapter 6 will describe the configuration of TOOLPACK project, chapter 7 will discuss the problems of installation on VAX/VMS. Chapter 8 will describe the actual capabilities of TOOLPACK project. Chapter 9 will compare the needs, goals, and capabilities described in the preceding chapters.

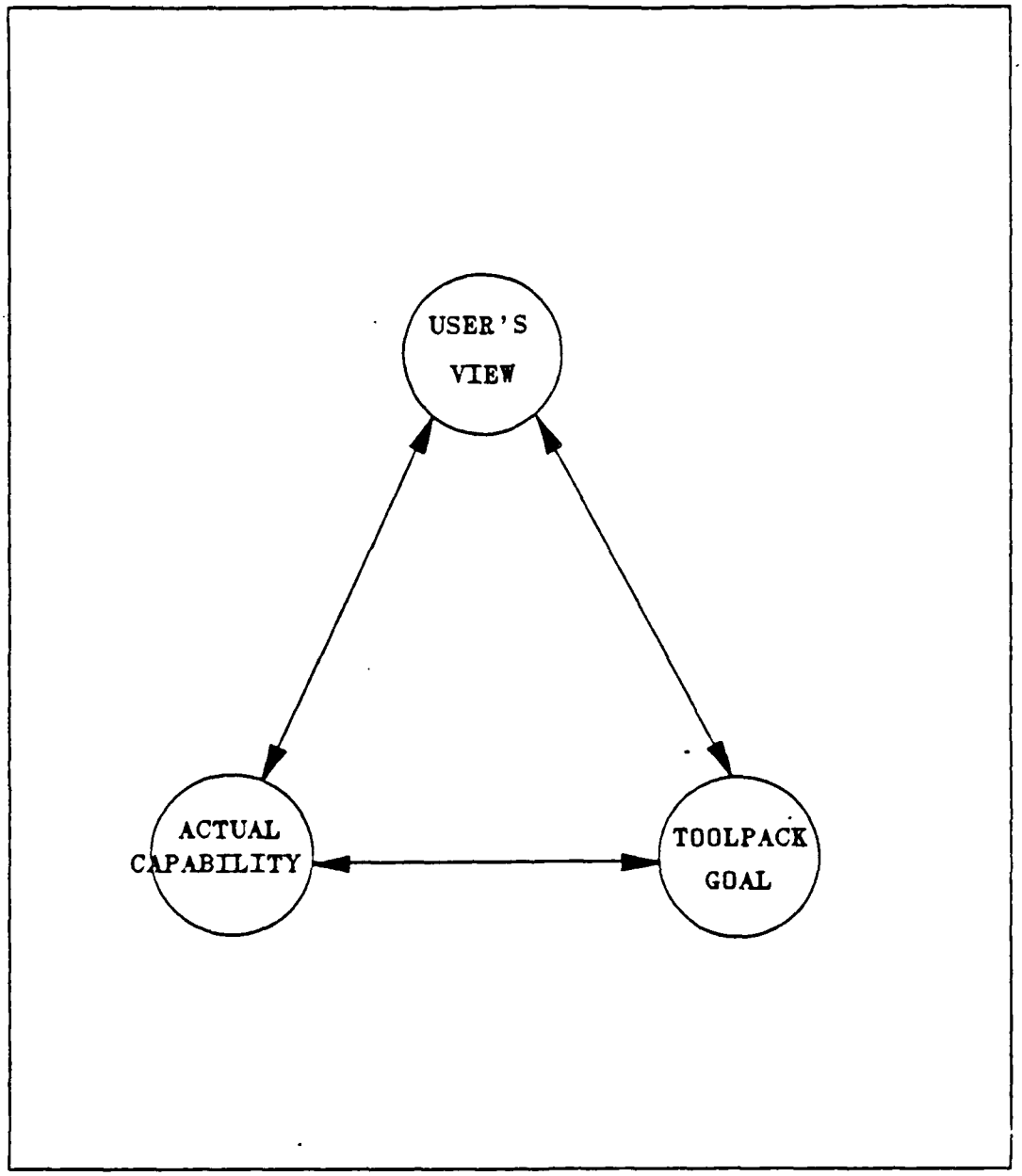


Figure 3.1 The Comparison Aspects.

IV. GOALS FOR TOOLPACK

From the workshop of the TOOLPACK project team came the following outline of the capabilities that TOOLPACK could provide to Fortran programmers [Ref. 1: pp. 5-6]:

1) A structured Fortran language which enhances standard Fortran with modern control and data structures. Such a language can contribute substantially to portability by permitting the use of "environment parameters" (such as the host machine's precision and overflow limit) and by permitting the generation of both single-precision and double-precision Fortran from the same version of the program.

2) Fortran program template processors which facilitate the production of Fortran preprocessors for problem statement languages.

3) Static data-flow analysis. A tool based on DAVE [Ref. 6] can detect such data-flow anomalies as variables that are assigned values but never referenced, variables that are referenced before being defined, and variables whose values may depend on whether the values of local variables are retained between subroutine calls. The static analysis capability could include a way of testing executability of paths in numerical programs.

4) Instrumention of Fortran programs with code to monitor execution characteristics.

5) Formatting of Fortran text.

6) Precision type conversion (double to single and single to double), for application to existing programs.

7) Conversion of standard Fortran to structured Fortran with automatic selection of appropriate control structures.

8) A general-purpose source to source transformation system in the spirit of TAMPR [Ref. 5: pp. 542-546].

9) A text editor with built-in knowledge of Fortran syntax.

Besides requiring integration of the above capabilities, the pack tradition demands integration of the software that supplies the capabilities. For that reason, the highest-level tools should rest on :

10) A common base of flexible components, including lexical analyzers,parsers,table managers,and report generators.

A summary statement of the goals is continued in [Ref. 19: p. 85]:

- The first is to provide a suite of tools to aid the Fortran programmer in the production and maintenance of medium-sized mathematical software projects.
- The second is to investigate the development of extensible programming support environments built around integrated tool suites.

The TOOLPACK architectural design document of 1982 [Ref. 7: p. 3] contains a list of objectives for the project. Ordinarily one would not expect any additional goals to be included in such a document. Portability (see 5, below) was elevated from a factor in the original project to a full status as a goal. As we shall see in the discussion, this decision to make portability an explicit goal has had a major impact on the design of TOOLPACK and on the utility of TOOLPACK:

1) The mathematical software whose production, testing, transportation and analysis will be supported by TOOLPACK is to be written in a dialect of Fortran-77. This dialect is to be carefully chosen to span the needs of as broad and numerous a user community as is practical.
2) TOOLPACK is to provide cost effective support for the production by up to 3 programmers of programs whose length is up to 5000 lines of source text. It may be less effective in supporting larger projects.
3) TOOLPACK is to provide cost effective support for the analysis and transporting of programs whose length is up to 10,000 lines of source text. It may be less effective in supporting larger projects.
4) TOOLPACK is to support users working in either batch or interactive mode, but may better support interactive use.
5) TOOLPACK is to be highly portable, making only weak assumptions about its operating environment. It will be designed, however, to make effective use of large amounts of primary and secondary memory, whenever these resources can be made available.

V. MATHEMATICAL SOFTWARE : CHARACTERISTICS, PROGRAMMERS, AND PROGRAMMING ENVIRONMENTS

This chapter discusses the current state and foreseeable evolutions in software development for scientific, mathematical, and engineering applications. In evaluating software support it is critical to identify the target user group and to characterize their background, jobs, work conditions and attitudes. Consideration of the factors that are likely and unlikely to change in the foreseeable future will make it possible to identify the feasible opportunities to improve software development for mathematical software.

It is widely thought that there will be a steady growth in the size and complexity of mathematical software. This growth will present new problems and with them new opportunities to improve programmer productivity.

A. DEFINITIONS

The central focus of software for scientific, mathematical, and engineering applications is numerical computation. We want to define numerical computation rather precisely so as to distinguish it from business data processing, symbolic processing (such as compilers), and general utilities (such as file manipulation systems or job schedulers) [Ref. 8: p. 687]:

> Numerical computation involves real numbers with procedures at a mathematical level of trigonometry, college algebra, linear algebra or higher.

Some people use a somewhat narrower definition which restricts the term to computation in the physical sciences and a few people even think of numerical computation as research and development computation (as opposed to production) in science.

Another definition has been suggested by Wayne R. Cowell [Ref. 4: p. 37]:

> Since about 1970, the term 'mathematical software' has been understood to mean computer programs that perform the basic mathematical computations of science and engineering. Someone characterizes the effort to produce mathematical software as the building of bridges between the numerical analysts who devise algorithms and the computer users who need efficient, reliable implementations of those algorithms.

In what follows we use the term "mathematical software" to refer to software for scientific, mathematical, and engineering applications and "programmer" for those people that write this type of software.

B. PROGRAMMERS

Mathematical software is primarily written by people with little or no formal training in computer science or computer architecture. These people have training in science, engineering or mathematics and call themselves physicists, engineers, etc, rather than the less prestigious "programmer". Although many of them spend the majority of their time doing programming, they regard the computer and programming to be a tool that they use in the work in their basic field. Throughout their careers these people will retain their primary allegiance to the basic field of science, mathematics, engineering that they were trained in.

There are relatively few computer scientists doing this kind of work because extensive training and experience in the area of application is required. In general, it is easier to teach scientists, engineers and mathematicians programming than to teach computer scientists the basics of an area of application. There is a group of numerical analysts and computer scientists whose major interest is in mathematical software. Their contribution has been significant and influential but they are and will remain a tiny minority of the people developing mathematical software.

Our characterization of the majority of programmers who work on mathematical software is that they have a mastery of the basic aspects of Fortran, a sophisticated view of algorithms and performance analysis for programs in their domain of interest, and a knowledge of structured programming. But they are not aware of basic concepts in algorithms to do symbol manipulation, language translation, and compiler theory nor are they aware of contemporary software development methodologies. For example, the concept, design, and coding of a program that has a computer program as it input data is beyond scope of their knowledge.

Since these programmers view computers and programming as a tool to do their work, they are very reluctant to learn new computer science ideas, concepts, algorithms, and techniques unless they are absolutely convinced that immediate and significant benefits will follow. Although they are very interested in innovative topics in their basic field, they are very conservative in their approach to new programming topics. Also their lack of basic training in contemporary computer science makes it difficult for them to learn new topics by reading the computer science literature.

The ignorance of computer science of most people that write mathematical software is equaled by the ignorance of most computer scientists about numerical computation. Many sophisticated scientists produce naive software just as many sophisticated computer scientists produce naive science.

Another important fact is that most mathematical software is written by a single programmer (or a small team). Thus there has been little pressure for program standardization. Also most mathematical software is maintained by the person who wrote it, so there is no compelling reason to develop documentation standards.

Most programmers in this area spend their entire professional career working in a single area of application. They

therefore have little opportunity to see software developed in other areas. This limited view of software together with an absence of a literature to present mathematical software has lead to their limited view of software development.

C. LANGUAGE

The most obvious feature of scientific programs is the language in which they are written : an overwhelming majority are written in Fortran. Some competition has come from PL/I and Pascal, the latter being popular in some circles especially for the implementation of prototypes, "quick and dirty" versions, etc.; both, however, remain marginal.

Many sites have done some experiments with Pascal in order to assess the fashionable language, but few have used it on actual projects, since most scientific programmers who have tried it deeply resent the lack of features they consider essential. Minor criticisms are of the absence of exponentiation, separate compilation and the inefficiency of run time checking. A major impediment has been the strong typing of arrays that includes their dimension , this has made it impossible to construct general purpose procedures to do array manipulations. Although the ISO standard now allows a remedy, it is still not widely available in Pascal compilers.

So Fortran is still king. It should be noted, however, that the world is not so simple as it used to be : Fortran means different things to different people. The Fortran 77 standard has not completely taken over; it many cases, what is available is still either a compiler based on the 66 standard, usually complemented by machine-dependent extensions, or some hybrid between the 66 and 77 versions. At the same time, some manufacturers are taking (high-risk) bets on the next standard being concocted by ANSI, referred to as Fortran 8X.

When talking about Fortran with respect to mathematical software, it is impossible not to mention an apparent paradox: in spite of its almost undisputed position as a vehicle for writing numerical software and its pretentions to portability, Fortran does not as yet offer any tool for controlling the numerical accuracy of programs in a portable fashion.

D. APPLICATION DOMAIN AND NEED FOR EFFICIENCY

The application domains of mathematical software are matrix calculations, linear systems analysis, and simulation of given engineering conditions in narrow applications. The most complicated application domains are control systems of nuclear power plants. These kind of applications need high reliability. They also need high quality (i.e., optimization) compilers.

There are two principle sources of the problems in the application of mathematical software: mathematical models of the physical world and the optimization of models of the organization world. The scope and range of the sources and the associated software is illustrated by the following list [Ref. 8: pp. 688-689]:

1) Simulation of the effects of multiple explosions. The software is a very complex program of perhaps 20,000 Fortran statements. It is specially tailored to this problem and may have taken several years to implement. The program requires all the memory and many hours of time on the largest and fastest computers.

2) Optimization of feed mixtures for a chicken farmer. This is standard software of modest length (500-2000 statements) even with an interface for a naive user. It might take substantial time to execute on a small computer (for example, mini-computer and personal computer).

3) Analysis of the structural vibration of vehicle. The software is similar to that of example 1. More computer time and memory would be used by this approach.

4) Simple linear regression on demographic data. This is standard software, but classical algorithms are neither reliable nor robust. Modern algorithms are short (200-400 statements) and execute quickly except for exceptionally large data sets.

5) Optimization of design parameters of a gyroscope. A mathematical model of a complex physical system is required and then optimization algorithms are applied. Determination of the gyroscope performance for a single set of parameters might involve the solution of a system of partial differential equations. Considerable human interaction is probably used to avoid astronomical computer costs and yet achieve some reasonable progress toward the optimum.

6) Calculation of the capacity of the wing tank of a jet liner. This is a simple problem except for the complex geometry of the wing tank. Once the wing tank is broken into simple pieces (probably by a person) then standard algorithms are reliable, short and efficient. The automatic processing of the complex shape requires much more sophisticated software of moderate size (perhaps 2000 statements), but still gives a short calculation.

The demand for efficient compiled code has lead manufacturers to produce sophisticated optimized compilers. These compilers are expensive to build and maintain and have achieved some of their efficiency by developing manufacturer unique extensions to Fortran that exploit unique characteristics of the hardware.

The demand for efficiency has lead to a reliance on high quality compilers for Fortran with extensions. This has in turn lead to major portability problems with Fortran.

E. PROGRAMS

1. Size

Scientific programs vary considerably in size. A typical range is between 2,000 and 50,000 source lines (whether or not one counts comments usually has a marginal influence on the evaluation). There are bigger programs, but they are not so common; some packages reach 300,000 lines or more, but one seldom hears about sizes comparable to what is often quoted about e.g., telephone exchange software (500,000 to million or more). Thus much mathematical software can be characterized as "medium-size".

There are many signs, however, that these figures may growing steady. The pressures toward larger programs

include : more extensive computations, more "user friendly interfaces", more error checking of input data and results, and more extensive reports and high quality display of results. This tendency is likely to bring about much concern regarding the scaling up of the methods used for program writing and project management.

In other areas of computer science the growth in the size of programs has lead to new problems ; among these are the problems associated with having more people involved in the development and maintenance of the programs. Also the development of programs for use by many people at many different locations presents new difficulties with training, documentation and maintenance.

2. Contents

We outline below some of the characteristics of mathematical and engineering software as we perceive them and as they distinguish this type of software from others such as business software (accounting, transaction processing and the like), real-time software (command/ control etc.), systems software (compilers, operating systems, teleprocessing etc.), or office information systems. These characteristics relate to the form and contents of the programs and to the way they are produced and used.

There is still a widely held view that mathematical programs are essentially computation-oriented. In our experience, this is inaccurate. Of course, most mathematical programs include some non-trivial arithmetic computation. If, however, one looks at the actual code, one frequently finds out that the part which actually performs numerical computation is relatively small in size (if not in execution time), the bulk of the program text being concerned with manipulation of data structures, storage management, input and output, pre- and post-processing,

error identification and exception handling, etc. In a large part, mathematical programs are data manipulation programs. In most cases, this part is growing much faster than the purely numerical one, which is often relatively stabilized; many developments have to do with improvements in the user interfaces, inclusion of interactive facilities, graphical input and output, uses of data base managements system, etc.

This aspect of mathematical programs should be understood by those who design new machine architectures, programming languages, software tools or methods aimed at this area.

3. Use of Extensions

Most programmers are aware of the non-standard Fortran extensions on their computers and recognize that their use restricts portability of programs. Despite this, most programmers feel that the use of non-standard features is necessary to achieve the maximum run time efficiency.

4. Programming Style

Mathematical software does not in general have a consistent program style. There is not any standard or commonly used programming style in mathematical software, most programmers have developed a unique approach that is not consistent with any contemporary approaches to software development (for example, top down design, information hiding, stepwise refinement). In addition, most programs do not display any consistent approach to program organization.

F. DEVELOPMENT ENVIRONMENT

1. The Program Library Concept

The idea of building a library of modules that could be reused in other programs was an early concept in Fortran software development. It is still very important. This concept has been surprisingly difficult to bring to fruition in the same sense as a library of books. That is to say,

widely available and good quality libraries for basic mathematical procedures did not become available until the 1970's and even now most computer users lack access to a good library of programs for mathematical software. This is in spite of expensive efforts by IBM and other computer manufacturers.

The author would like to classify mathematical software libraries into three types :

1) Low level (utility function) library.

The libraries of this type are basic mathematics, trigometrical functions and are widely used in mathematical programming. The illustrations of this type library are SQRT (square root), SIN, COS, and EXP, etc.

2) Middle level library.

The libraries of this type do simple mathematical, statistical, and graphical functions. The algorithms of the programs are simple and easy to understand. Most libraries are used independently by a single user and do not support an integrated scheme. Most users could write and modify these programs. The illustrations of this type are IMSL (International Mathematical and Statical Libraries, Houston, Texas) and NAG (Numerical Algorithms Group, Oxford, England), etc.

3) High level library.

The libraries of this type are large specialized application packages which provide integrated algorithm schemes. The vast majority of the users of these packages understand the action of the algorithms but do not know the coding details. The examples of this type are LINPACK, EISPACK (a systematized collection of programs for eigenvalue problems), and NASTRAN (a structural engineering package), etc.

The library concept is based on the fact that many problems are of a somewhat standard nature and occur in many different contexts. This is especially true of numerical

widely available and good quality libraries for basic mathematical procedures did not become available until the 1970's and even now most computer users lack access to a good library of programs for mathematical software. This is in spite of expensive efforts by IBM and other computer manufacturers.

The author would like to classify mathematical software libraries into three types :

1) Low level (utility function) library.

The libraries of this type are basic mathematics, trigometrical functions and are widely used in mathematical programming. The illustrations of this type library are SQRT (square root), SIN, COS, and EXP, etc.

2) Middle level library.

The libraries of this type do simple mathematical, statistical, and graphical functions. The algorithms of the programs are simple and easy to understand. Most libraries are used independently by a single user and do not support an integrated scheme. Most users could write and modify these programs. The illustrations of this type are IMSL (International Mathematical and Statical Libraries, Houston, Texas) and NAG (Numerical Algorithms Group, Oxford, England), etc.

3) High level library.

The libraries of this type are large specialized application packages which provide integrated algorithm schemes. The vast majority of the users of these packages understand the action of the algorithms but do not know the coding details. The examples of this type are LINPACK, EISPACK (a systematized collection of programs for eigenvalue problems), and NASTRAN (a structural engineering package), etc.

The library concept is based on the fact that many problems are of a somewhat standard nature and occur in many different contexts. This is especially true of numerical

computation because scientists and engineers use the language of mathematics in their analysis.

The methods one uses seem to be independent of the particular computer and thus expressible in some machine-independent Fortran subset. Fortran, Algol and their descendents have made it possible to attempt to develop the science, art and body of numerical computation software. Even with these advances, the significant differences among compilers has hindered progress.

2. Tools

The use of programming tools, beyond such standard ones as editors and compilers, is fairly limited in many installations. It is remarkable to see, for example, how often the machine-format dump still plays the role of the basic debugging aid. Here again, the discrepancy in levels of abstraction between the sophistication of the applications and the people who conceive them , on the one hand, and the characteristics of the underlying software, on the other hand, are striking. Also, one can again notice the negative effect of the language : although Fortran is much more primitive by its concepts than, say, Pascal or Lisp, it is often less amenable to language dependent tools such as syntax-directed editors, symbolic debuggers etc. because of its baroque features, strange format and irregular structure.

Mathematical software has also been the prime target for other successful tools: Fortran static (and, to a lesser extent, dynamic) analyzers. Again, these tools are underused; it is clear, however, that they can provide a host of services which, although conceptually limited, are extremely useful in connection with the development, acquisition, debugging and documentation of mathematical software.

Although it is true that some of the checks performed by Fortran static analyzers (for example, type

checking) are only needed because of the language's deficiencies, this is only part of the picture; some of the ideas could be profitably adapted to more elaborate languages, which are still lagging behind Fortran with respect to availability of such tools.

G. PORTABILITY

While everyone recognizes the potential savings from distributing good software, it has been hard to achieve even when good, usable software is written. The dependency of mathematical software on machine word length as well as the eccentricity of compilers and operating systems pose formidable barriers to the dissemination of quality software.

It has been shown that portability and top efficiency cannot be achieved simultaneously in a high level language like Fortran because of compiler variations. A 100 % loss in efficiency may be an acceptable price to pay for portability in some instances, but there are even more severe problems with error handling, precision changes (double precision to single precision or vice versa), and arithmetic unit behavior. These difficulties should be isolated and methods found to overcome them in an automated system.

One method to obtain program portability is to define a standard widely accepted language and then write preprocessors that translate programs written in it to a language for which a good compiler exists. The most notable such effort in mathematical software is the RATFOR (Rational Fortran) language [Ref. 9: pp. 285-318] that extends Fortran and is transformed into Fortran. There have been several problems with this approach :

1) The preprocessors are somewhat difficult to write and they also must be modified to keep current with the Fortran on the target machine.

2) The resulting Fortran is very hard to read and thus it is difficult for the programmer to modify or optimize the Fortran directly.

3) Any error messages are reported in terms of the Fortran program rather than the RATFOR program.

Another problem for portability is the spread of small machines such as micro-computers (personal computers) and mini-computers. These machines are so numerous that it is not possible nor economical to do a careful job on the mathematical software (which may be permanently implemented as micro-code).

The manufactures of such machines are frequently unaware of quality software principles and portability software for mathematical software. The result has been numerous instances of inadequate algorithms-both in the hardware and in manufacturer supplied libraries and systems (for example, the Fortran built-in functions).

H. HUMAN ASPECTS AND PROGRAM TECHNOLOGY

Everyone agrees that human engineering of software is important, but so few people do anything about it. There have been instances of mathematical software that was widely used because they had good human engineering even though the results computed were unreliable. .These and other experiences have convinced many (but far from most) developers of numerical software that the human engineering (user convenience) aspects are critical. This is, in itself is a milestone; unfortunately there have been few advances in how to do human engineering. It still seems to take a lot of hard, patient work.

Professionals in mathematical software have always had their favorite methods for various kinds of problems (there is not a general methodology for mathematical programming). Occasional surveys show that there is no consensus among the experts as to which methods are best. Even worse, for many years, most people did not distinguish between a somewhat vague method and a computer program implementation of the method. Now people realize that the implementation (software) is as critical as the method, as there can be (and have been) terribly poor implementations of good methods.

There are two main variables here [Ref. 8: p. 688]: different implementations of the same method and different methods for the same problem. It is not at all easy to design frameworks in which meaningful comparisons can be made. However, in the late 1960's such comparisons were started for ordinary differential equation software and now the framework for this particular area is well defined. Since then there have been significant accomplishments in evaluating software for numerical integration, special functions, linear algebra and polynomial root finding.

Finally, there is a notable lack in the use of "program proof methods" for software for mathematical software. Some reasons for this are :

- it is difficult to incorporate the uncertainties of round-off into proofs.
- the software tends to be too long for current proof methods
- most numerical computation software has parts whose performance cannot be specified in terms of input-output relationships

VI. CONFIGURATION OF TOOLPACK

A. THE TOOLPACK TOOL INTEGRATION CONCEPT.

The TOOLPACK tool integration concept is centered around the notion that software tools must be focussed on supporting the creation and deep understanding of a large and complex mass of information-namely the software under development. It seems that software workers often attempt to view their jobs in this way, but that they are most often thwarted in their attempts to exploit this view by system software.

Thus the TOOLPACK project is attempting to create a Portable Software System (PSS) which can be profitably and effectively viewed and operated as a system for the manipulation and management of the large, complex and multifaceted object which is the software program under development. This section will present a necessarily brief overview of the integration software which was developed for the TOOLPACK project. The entire system of tools and the encompassing integration software has been named the Integrated System of Tools (IST).

B. THE FILE SYSTEM.

As above indicated, the central focus of the TOOLPACK command language is the creation, accessing and maintenance of the data repository whose aim is the faithful and supportive represention of all the data which the user needs in order to perform needed software work. This represention seems to be effectively achievable by portraying the software are a structured, coordinated set of views and versions.

In TOOLPACK these various views and versions are all stored as files. Thus the heart of TOOLPACK is a file

system. This file system actually consists of two parts : the host filestore (HFS) and the portable filestore (PFS) [Ref. 14: p. 5].

HFS files are simply formatted files in the host system filestore. They are always accessible to host system utilities. The virtual machine makes no assumptions about an HFS directory structure and the only assumption about the names of HFS files is that they may not exceed maxpath-1 characters, which maxpath is a virtual machine constant set by the installer. To identify an HFS file, the user adds a one-character prefix (usually '#' but installation-dependent) to its name. This "host file-id" character is stripped off the name before it is used.

PFS files reside in a tree structured directory system similar to that provided by VMS system. A directory in the tree structure may contain files or other directories. Conceptually, the directories may be nested to any path, but there are practical limitations, such as the maximum length of name. PFS files may not be directly accessible to standard host system utilities.

In addition to these disk files there are four preconnected files available to tools. These files are available in all operating regimes. These files and their symbolic names are as follows [Ref. 14: p. 6]:

1) The standard input file ('stdin'), normally associated with the user's keyboard.
2) The standard output file ('stdout'), normally associated with the user's terminal.
3) The standard error reporting channel ('stderr'). The associated of this channel is host dependent but is often the user's terminal.
4) The standard list channel ('stdlst'). The association of this channel is host dependent for connection to a spooled system printer, though it may be connected to a fixed file that can be printed by the user.

Almost all files within the HFS and PFS are formatted sequential, but formatted direct access file capability is provided.

C. THE VIRTUAL MACHINE (TVM) OF TOOLPACK

To provide the capabilities of the TOOLPACK tools and command interpreters on as wide a range of machines as possible, a definition of the TVM was produced to which all TOOLPACK programs comply. The TVM includes definitions of the character set, directly structured and minimum machine capabilities that must be provided on the host computer before TOOLPACK programs can be used.

The TVM capabilities are actually accessed by the tools via the Tool Interface to the Environment (TIE) library. In order to make the capabilities of the TVM available, it is necessary to have an implementation of the TIE library on the host machine. A portable version of the TIE library, written in Fortran-77, is provided in the TOOLPACK software suite to allow as many users as possible to at least try out TOOLPACK capabilities.

> It is probable that a single redesign of the portability base will be undertaken in the light of experiences gained as a result of the production, distribution and use of the first release. If this does happen, great care will be taken to provide a support mechanism for all existing tools, with a compatible upgrade path. Any new design would be aimed at producing a smaller, more appropriate and more easily implementable core library of routines for the TIE [Ref. 19: p. 91].

The main features of the TVM are as follows :

1) A stream-based input/output system for files and preconnected units.
2) A fixed internal character set and variable-length string handling.
3) A tree based directory structure and defined file naming convention in a Portable Filestore (PFS), normally separate from the Host Filestore (HFS).
4) A defined process-scheduling and argument-passing capability.
5) Extensible capabilities by the use of supplementary libraries.

By splitting the TIE library into several sections it has been possible to allow greater environmental flexibility

for the use of tools. The split is shown in Figure 6.1, which shows that there are three sublibraries to the TIE; common, input/output, and flow-of-control.

The common sublibrary contains general-purpose routines to convert character types, manipulate strings and recover the date and time. There is only ever one version of this library in a TIE implementation.

The input/output sublibrary is concerned with the provision of the stream based input/output routines and the directory handling capabilities of the PFS. There may be two separate implementations of this sublibrary, one of which provides the full capabilities of the HFS and PFS, as defined for the TVM, and the other which maps all input/output and file access to host files and ignores directory handling routines. The two versions of the sublibrary contain the same routines with the same apparent (to the tool) functionality so no code changes are required to use either version.

The flow-of-control sublibrary is concerned with the initialization and termination of TOOLPACK tools, process scheduling and argument passage. There may also be two versions of this sublibrary, one which allows process scheduling and the other which does not.

The provision of multiple versions for some of the sublibraries allows for a variety of levels of implementation of the TIE and lets the tools be used in a variety of environments without modifications to the source code. This flexibility can be useful during the TIE implementation phase, during tool development and for those users who, while wanting access to the capabilities of TOOLPACK tools, are not interested in the provided command interpreters and PFS. The possible environments available are as follows [Ref. 13: pp. 3-4].

1) Embedded : This is the full environment defined for the TVM. Both the PFS and HFS are available and tools may be scheduled from TOOLPACK command interpreters.

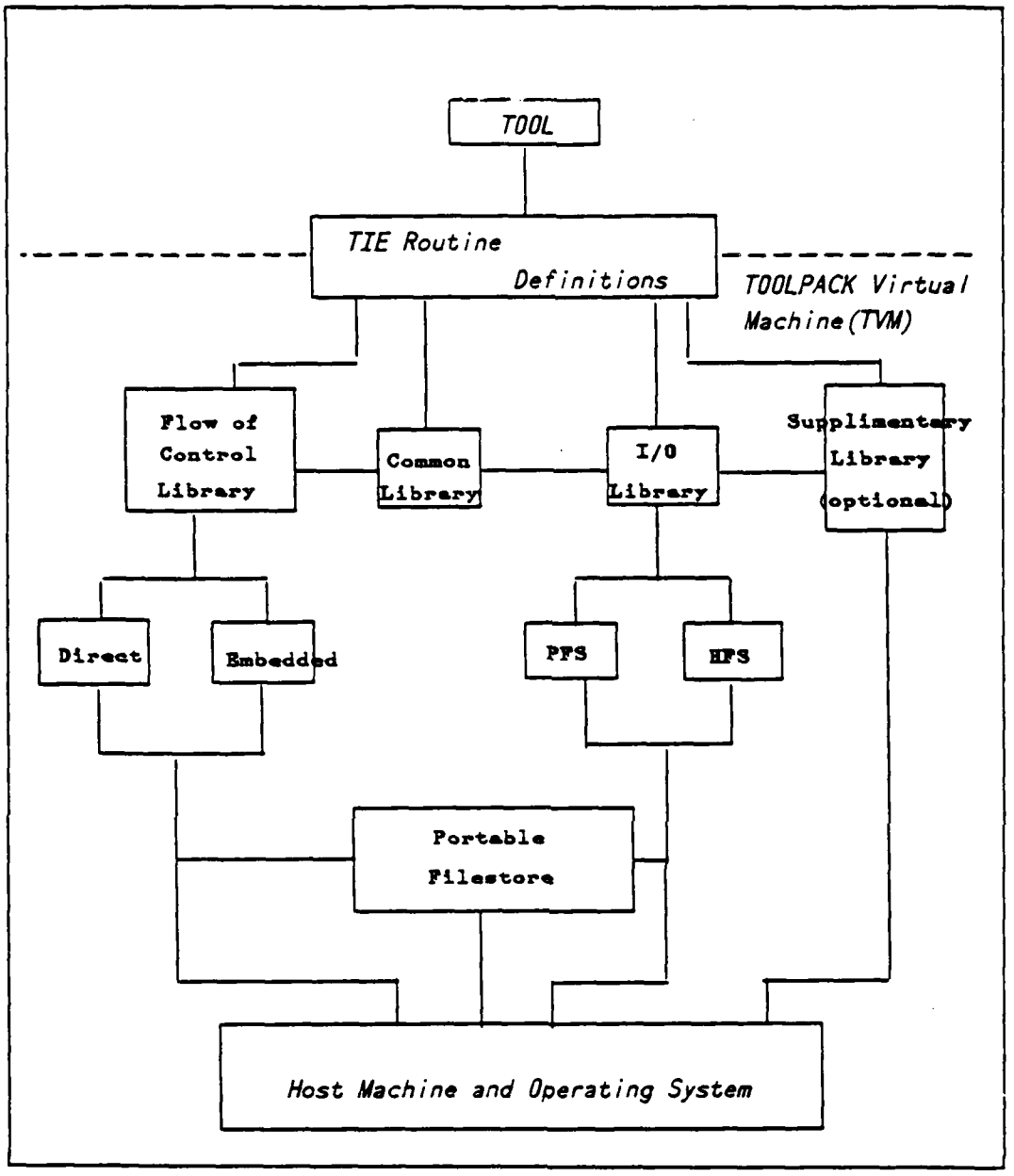


Figure 6.1 Sections of the TIE Library.

2) Stand-atop : This environment still allows tools to access both the PFS and the HFS but assumes that tool scheduling is performed direct from the host operating system. TOOLPACK command interpreters will not be able to schedule tools in this mode.

3) Stand-astride : This environment provides the same flow-of-control capabilities as the embedded mode but

access to the PFS is not possible; all I/O operations are mapped automatically to the HFS and directory management is not possible.

4) Stand-alone : In this mode, tool scheduling is performed direct from the host operating system as in stand-atop and the same file access restrictions apply as for stand-astride. (The installation of this thesis is stand-alone mode.)

The TVM definition can be extended in any area by the use of TIE Supplementary libraries. The current Supplementary library includes access functions for token streams and parse trees, extended string handling, pattern matching and data structure handling. The access functions provided allow easy access to, and manipulation of the lower level representations of program units. These lower level facilities greatly ease the production of custom Fortran manipulation tools by users; a tool writer may freely manipulate a token stream knowing that it can be generated for him and that the modified form can be easily returned to the source text.

Before TOOLPACK tools or command interpreters can be used it is necessary to have an implementation of the TIE library available. The TIE implementation may be available in the following ways :

1) The portable TIE implementation may be installed on the host.

2) A customized TIE implementation may be produced, either completely new or a modification of the portable version.

3) An available customized implementation may be used.

While it may seem that providing an implementation of the TIE is a lot of work, it is likely to take less effort than mounting a comparable suite of monolithic tools that do not conform to a portability base.

VII. INSTALLATION ON VAX/VMS

The installation of TOOLPACK is a very difficult task. On the VAX/VMS 11/780 VMS system it took several months to get the most basic "STAND-ALONE" mode up and running. The documentation is voluminous, difficult to read and the material is not presented in the same order as the steps that are necessary for installation. Without additional help that was found only after numerous phone calls around the country, it is not possible to successfully install the VAX version. As noted in the documentation, the NAG office can not (and other than providing user telephone numbers did not) provide help with the installation. The help of Larry Reed of the National Energy Software Center was pivotal in getting the installation completed. The author is indebted to him for the successful installation.

The expenditure of resources was quite large. The author worked for several weeks reading the documents. It was then determined that the task was beyond the capabilities of someone who was not an experienced VAX systems programmer. Andrea McDonald, a systems programmer on the staff of the Computer Science Development, then worked on the installation on and off for 2 months. During this period all the files were established, corrections posted, programs compiled and libraries established. It later was found that much of this work was unnecessary to set up the VAX/VMS STAND-ALONE mode. Finally with several calls to Larry Reed and the simultaneous help of Andrea McDonald and my thesis advisor both working full time for two and one half weeks we were able to get several of the basic tools working.

By the time the system was finally installed, a new version of TOOLPACK had been released. It was decided to start the installation completely from the beginning and

install version 1.4. The VMS/TIE installation took one week (60 hours plus) of full time work to establish the file system, read in the files, post and document in the code all the corrections for 1.2, 1.3 and 1.4, and establish the libraries. An additional week was needed to post the corrections in the individual tools, compile the tools, test the tools and build VMS EXECS to execute the tools. This two weeks of work was begun immediately after the first installation, so the full knowledges of that installation allowed the second installation to be done in the minimum possible time.

The author's advisor documented the second installation so there would be a permanent record of the steps necessary to install the VAX/VMS STAND-ALONE version of TOOLPACK. That description is now in draft form; a decision on publishing it as a technical report is awaiting TOOLPACK version 2. It is likely that after modifications to accommodate version 2 it will be published as a technical report. The present draft is over 30 pages.

VIII. CAPABILITIES OF SYSTEM AND TOOLS

A. GENERAL.

There are basically two major classes of methodologies: Single_module testing, and multiple-module testing (or system testing). The goal of this section is to explain a base line of understanding about the processes that apply in typical quality assurance circumstances.

A way of determining how to work a testing methodology is to examine the characteristics of various methods against a backdrop of some typical situations. The four cases we can employ for this purpose are:

1. Case a : A high-criticality single module.
2. Case b : A medium-sized, medium-criticality system.
3. Case c : A large, medium-criticality system.
4. Case d : A large, high-criticality system.

Case a is in a high-level language, it is may be 250 to 500 statements long, and it has a very complicated control structure. It has many different things to do, and it must be error free.

Case b is a set of about 25 modules that support an on-line facility of some kind. If the software system fails, there is no serious loss because there is an automated back up system; however, it is expensive in terms of lost production and associated waste. The system is written in a structured extension of Fortran and containes about 6,500 statements overall. The calling depth in the structure of the system is between 4 and 7 - not complex and not flat.

Case c is a comprehensive system for control of a facility, and it is 80 % in a high-level language like Pascal, Fortran, or Ada and 20 % in an assembly language. The total volume of code is in the range of 175,000 lines of code. If this system fails, there is a substantial loss of

value, but no lives would be lost, and the damage would not normally be extensive and/or expensive to repair.

Case d is a geographically dispersed interactive control system where human life is at stake-like an advanced air traffic control system. The system approaches the limits of current complexity in the sense that the latest methods are employed in its design and implementation, perhaps 1,000,000 lines of code overall [Ref. 11: p. 376].

We would classify the TOOLPACK into multiple_module and Case c (A large, medium-criticality system).

Another software system evaluation methodology is benchmark testing. Benchmark testing is the application of synthetic programs to emulate the actual processing work handled by a computer system. Benchmark programs permit the submission of a mix of jobs that are representative of the users' projected work load. They also demonstrate data storage amounts and provide the opportunity to test specific functions performed by the software system. Through this technique, the limitations of the system become apparent early in the acquisition process. Sometimes user organizations will insist that the results are attached to software system specifications, formally stating that a specific number of transactions can be processed in a given period of time, the response to an inquiry will be within a stated amount of time, and so forth.

Benchmarks can be run in virtually any type of system environment, including batch and on-line job streams, and with the users linked to the system directly or through telecommunication methods. Common benchmarks are the speed of the central processor, with typical programs executed in a set of subprograms, as well as multiple streams of jobs in a multiprogramming environment. The same benchmark run on several different computers will make speed and performance differences attributable to the central processor apparent.

Benchmarks also may be centered around an expected language mix for the programs that will be run, a mix of different types of programs, and applications having widely varying input and output volumes and requirements. The response time for sending and receiving data from terminals is an additional benchmark for the comparison of systems. Sometimes, rather than running actual benchmark jobs on computer system, system simulators are used to determine performance differences. In commercial systems simulators, the workload of a system is defined in such terms as : how many input and output operations there are, how many instructions are utilized in a computation, and the order in which work is processed. The specifications are fed into a simulator that stores data about the characteristics of particular equipment (such as instruction speed, channel capacity, and read-write times). The simulator in turn processes the data against the operating characteristics and prepares a report of the expected results as if the actual computer were used. Then the systems characteristics can be changed to mimic another model of computer and a new set of performance data produced for comparison. The time and expense of running actual benchmark programs on a computer is of concern to analysts and specification alike. Thus, the use of commercial simulators is an attractive alternative [Ref. 12: pp. 586-588].

B. EVALUATION AND OPERATIONS PROCEDURE UNDER VAX/VMS

Up to present, we have discussed the general principles of evaluation techniques. I selected the evaluation method benchmark testing for this thesis. As mentioned above, the installation of TIECODE is not a Embedded regime, but it is a STAND-ALONE environment.

Therefore, the STAND-ALONE regimes allows the tool writer to make use of the TIE library without the need for either the IST (Integrated System of Tools) command

interpreter (ISTCE) or the portable file system(PFS). In this regime, all file I/O is directed to host files regardless of the manner of the file access requested. Tools operating in this regime are executed directly from the host operating system, to which control is returned on completion [Ref. 13: pp. 3-5]. The appendix A of TOOLPACK/1-Introductory Guide [Ref. 14: pp. 1-10] has examples for the Embedded regime, is not for the STAND-ALONE regime. Therefore, the author followed the procedures of Tutorial Examples of the TOOLPACK/1-Introductory Guide, but some procedures do not work in the STAND-ALONE regime, as will be pointed out in following sections. Some selected TOOLS, for example ISTPL and ISTPT used the benchmark testing technique (time comparison with different size programs).

For the benchmark testing, the author produced the programs which are a modifications of a Command program under the VAX/VMS System (Version 4.2). This program is attached in Appendix A.

The TOOLPACK Manual has the descriptions of the input parameters and the output parameters which are the parameters in each tool. But this kind of expression is not very helpful to the users. Therefore, this thesis uses the D.F.D. (Data Flow Diagrams) conventions which are more understandable, readable, and extensible than TOOLPACK's [Ref. 17: pp. 47-124].

Especially, the D.F.D. conventions are very useful to represent a lot of parameters which are combined for more than two tools.

1. ISTLX (Fortran-77 scanner).

a. Description

ISTLX is a Fortran-77 scanner that converts Fortran-77 source text to a token stream and detects and reports lexical errors. The scanner has been mechanically

generated from a specification of the Fortran-77 language [Ref. 15: p. 1].

ISTLX reads Fortran-77 source text from the source file. The resulting token stream is placed in the token file and the comments are placed in the comments file. Any errors discovered are reported to the error file and an attempt is made to continue scanning by deleting or adding tokens. During operation the scanner produces a list file which contains the input source text preceded by the token number of the first token for each statement.

ISTLX is the first step in using each tool. Therefore, if ISTLX produces an error file, then the user should correct the source text programs which are Fortran-77 source text.

There are a lot of users who are using the WATFIV source program [Ref. 16: pp. 65-101] at Naval Postgraduate School. The user must be careful with the control structures of WATFIV programs which are not admitted by TOOLPACK. Figure 8.1 shows an example.

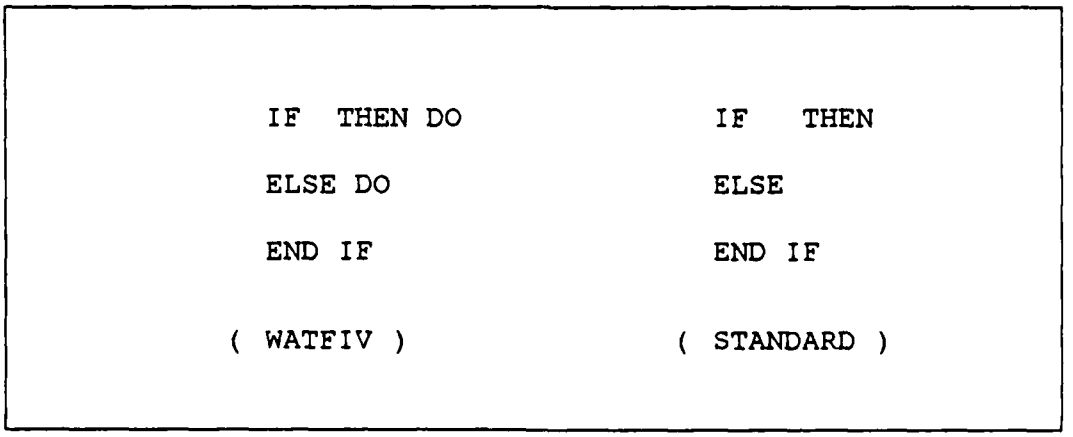


Figure 8.1 WATFIV control structure.

b. The Flow of ISTLX

Figure 8.2 shows the parameters using the D.F.D.

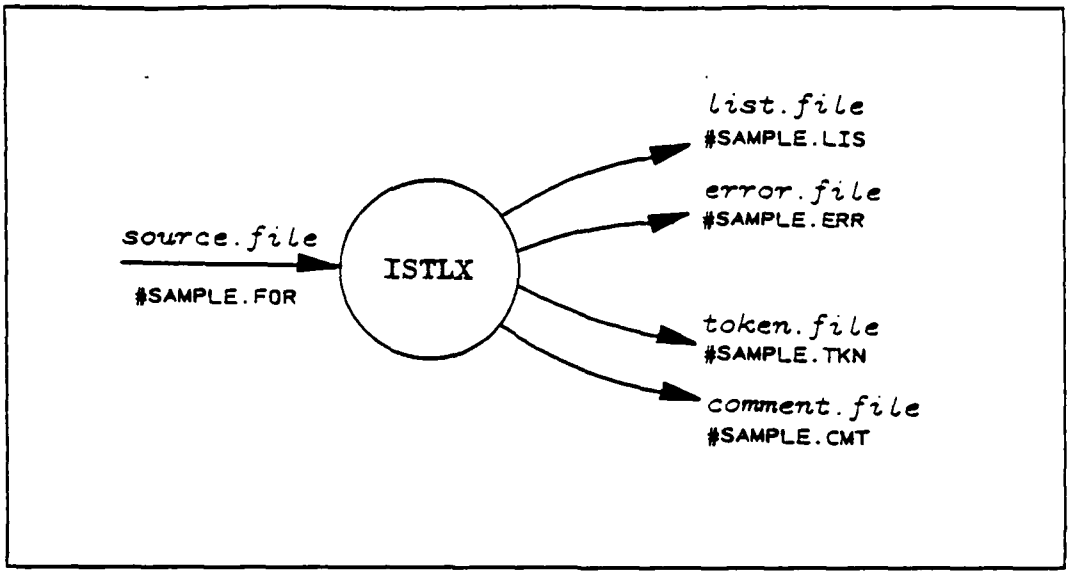


Figure 8.2 The Flows of ISTLX.

c. CPU Time Comparison

The CPU Time comparison of different size programs is shown in TABLE I.

TABLE I
CPU TIME COMPARISON OF ISTLX

(unit : secs)

program size	85 LOC	700 LOC	1500 LOC
1	6.91	59.70	98.06
2	7.01	59.18	97.05
3	7.00	59.55	97.66

2. ISTYP (TOOLPACK Parser)

a. Description

ISTYP parses a Fortran-77 program. It takes as its input a token stream produced by ISTLX (Fortran-77

Scanner) and produces a parse tree, symbol table and comment index [Ref. 18: p. 3].

All error and warning messages produced by ISTYP are written both to the standard error channel and the symbol table file. When a tool which uses the symbol table is executed, these warning and error messages are displayed again. As many error conditions render at least part of the symbol table or parse tree information invalid, it is important that the user is aware of the possiblity that further processing may be completly useless.

b. The Flows of ISTYP

To execute the ISTYP, the previous step which is the running the ISTLX is required. Therefore, the users would follow the Figure 8.3 (The Flows of ISTYP). The command file which is to execute the ISTYP is Appendix B.

c. The Problems Using The ISTYP in the VAX/VMS System

In chapter 7, the author already mentioned the problems associated with the installation of the TOOLPACK. The version of the distributed *TOOLPACK* is TOOLPACK/1, Version 1.4. But, this version stills need a lot of corrections.

The TOOLPACK programs are composed of many modules. Therefore, if we correct some modules then we have to correct the related modules. These kind of jobs are not easy. If the installer is not careful in identifying and applying all the relevant corrections, then errors arise that are very difficult to trace. The difficulties in installing and maintaining the programs seem all the more ironic since the aims of the TOOLPACK project were to provide a suite of tools to aid the Fortran-77 programmer in the production and maintenance of medium sized mathematical software projects [Ref. 19: p. 85].

One example of the problems encountered was the attempt by the author to apply TOOLPACK to a 1500 line

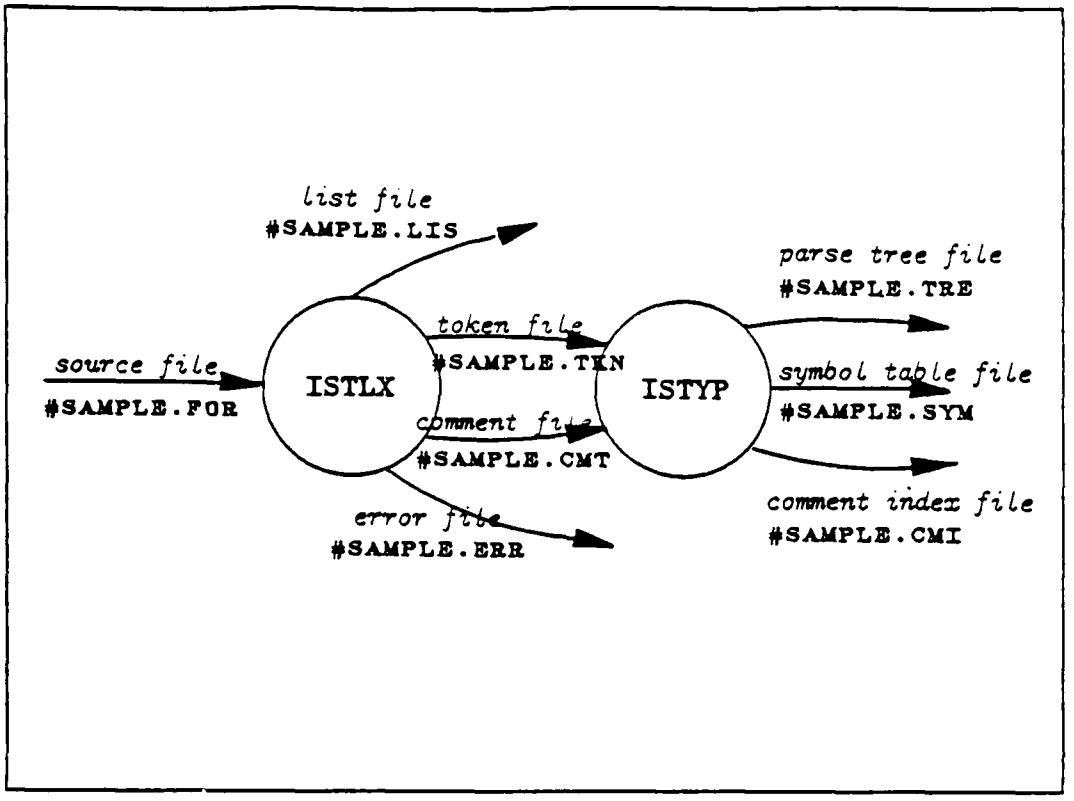


Figure 8.3 The Flow of ISTYP.

Fortran 77 program. The program was executed by ISTLX correctly, but it failed in ISTYP. The reason was that "string_size" is defined to 7500 in the YPDEFS of ACCESS FILES Directory (for example : define (string_size,7500)).

After the problem was found, the author changed the definition statement from "define (string_size,7500)" to "define (string_size,15000)". Then, the execution times of the ISTYP jumped to over 1.5 hours, eventually this program was terminated. Until the problem is corrected, the users of the Naval Postgraduate School version must limit its use to less than 1,000 lines of code.

3. ISTPL (Polishing Tool) /ISTPO (Option File Editor)

a. Description

ISTPL is a formater for programs written in Fortran-77. It takes a token stream produced by ISTLX, and produces a text file containing the formatted program. It is controlled by an option string from an option file (if the option file is not supplied then the program uses the default settings), together with any command-line options. The source program of Fortran-77 must be "lexically correct" in the sense that it may be analyzed by ISTLX without producing errors [Ref. 20: p. 3].

As ISTPL makes no use of the original source file, it may be used as an "unscan" program which turns the token stream into text. When ISTPL detects an error, it may sometime be able to recover from it and continue processing the user's program. In this case the error message will be inserted into the output file as a comment beginning with "C*PL*ERROR*" (see Appendix C).

There are two kind of flows to execute the ISTPL. One is the simple execution with the default settings. The time of the execution is faster than the others.

The complex flow shows the more sophisticated output possible with use of the option file. But the time of execution is longer. The usage of the methods is dependent on the users. The simple flow is shown by Figure 8.4 and the complex one is shown by Figure 8.5. The command files that are to execute the ISTPL are Appendix B.

b. Option File Specification

An option specification has the format "Parameter = Value". The actual options available are described in a later section. Options specified as ISTPL parameters 5+ (command-line options) will override option specifications from the option file. Because of the large

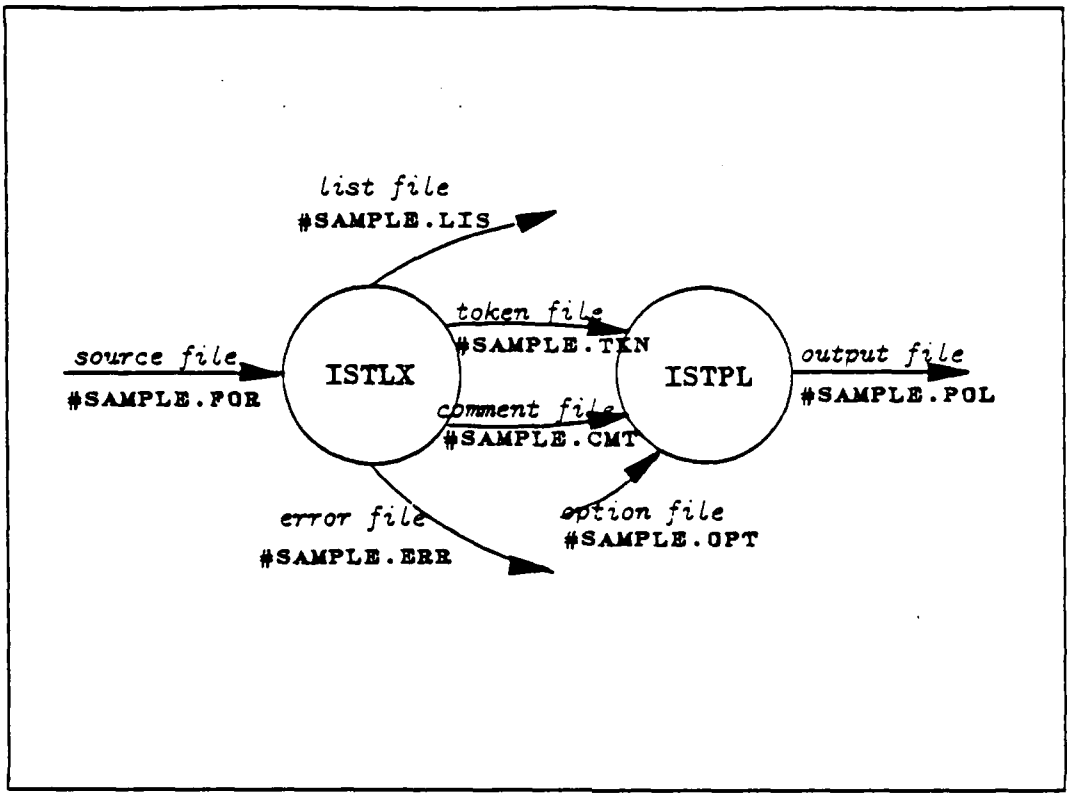


Figure 8.4 The Flow of the ISTPL (Simple Method).

number of possible options for ISTPL, an option file editor ISTPO is provided and is recommended for option file creation and maintenance.

c. Caution of Usage of ISTPL (with complex method)

As mentioned above, the complex method of running the ISTPL uses the ISTDS (a declaration standardizer that rebuilds the declarative parts of Fortran-77 program units according to a programmable template [Ref. 21: pp. 3-4].

Therefore, some errors do not occur in the Simple Method but in Complex Method. Sometimes, the authors of Fortran-77 programs do not define the names which are External Subroutine Names (An example is shown by Appendix C). Nevertheless, these programs work correctly under the given Fortran-77 Compiler [Ref. 22: pp. 1-4].

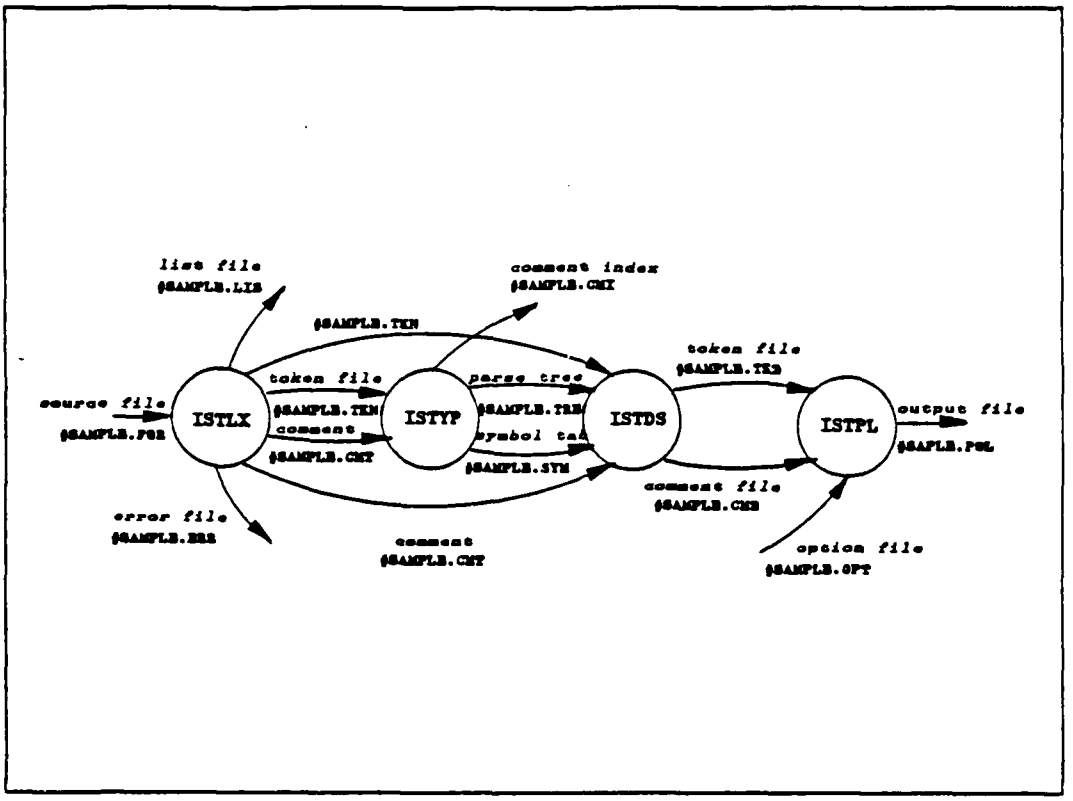


Figure 8.5 The Flow of the ISTPL (Complex Method).

But if the user wants to run these programs using ISTPL then the following errors are reported:

- C*PL*ERROR Unexpected statement type
- C*PL*ERROR Unexpected <TZEOS>
- C*PL*ERROR Internal Error (GRIND1)-TZEOS confusion

The reason was that ISTDS was confused whether the name of the subprogram was defined or not in the main program declaration part. When the user writes the function subprograms, tool ISTPL doesn't report this kind of error message.

Therefore, if the users of ISTPL have this kind of error, then they should correct the source program which is the Fortran-77 program.

d. ISTPO (Option File Editor)

The following are a summarization of the useful contents for the users from ISTPO users' guide [Ref. 20: p. 12]:

1) Description : ISTPO is an editor for ISTPL option files. It is menu-driven and has an inbuilt help facility [Ref. 20: p. 12]: The only parameter of the ISTPO is the Name of the option file (e.g., SAMPLE.OPT).

2) The operation of ISTPO : When ISTPO is started, it will attempt to read the option file specified. If it can not do this (e.g., because the file does not exist) ISTPO assumes that a new file is being created and all options are internally set to the ISTPL defaults. Options are changed by giving the ISTPL option specification as a command. Option names and values may be abbreviated so long as they remain unique, apart from token names which must be given in full.

3) Command Format : The ISTPO commands are : Exit, Help, Menu, Next, Query, Quit, Read, and Write. Commands may be abbreviated so long as the abbreviation is not ambiguous. Arguments to these commands should be on the same line as the command itself, separated by a space or spaces from the command name. Commands which require arguments (rather than having optional ones) will prompt for any missing arguments.

4) Command Descriptions.

- The EXIT command writes out the option file as modified and terminates ISTPO. There are three forms of the HELP command: "HELP", "HELP?", and "HELP topic". The first form displays the current ISTPO menu. The second form lists the topics on which help is available. The third form displays information about the requested topic.
- The MENU command moves to the specified ISTPO menu. If no menu name is specified following this command, then the current menu is redisplayed.
- The NEXT command advances to the ISTPO menu which follows the current one.
- The QUERY command toggles query mode, which is initially off. In query mode, all parameter changes are confirmed before being done.
- The QUIT command terminates ISTPO without writing the option file. This command is same VAX/VMS QUIT command in the EDITOR mode.
- The READ command reads an option file. This will completely replace the current state of the option memory.
- The WRITE command writes the option file but doesn't terminate ISTPO. If a file name is specified after the command, then that file becomes the option file being edited.

5) Menus : There are 8 menues in ISTPO. These are DIR, BASIC, COMMON, UNCOMMON, BLANK_LINES, LINE_BREAK, SPACING1, and SPACING2. Initially the user is at menu DIR, which consists of a list of the other menus and simple operating information. There are many options for the ISTPO BASIC menu [Ref. 20: pp. 12-13].

The author selected some of them to demonstrate the results which are shown in Appendix C. The followings are the selected BASIC options :

- INDIF = 10 ; (Indentation within a block_IF)
- IOTHCO = .TRUE. ; (Insert CONTINUE statement before previously labelled executable statement)
- LMARGS = 7 ; (Left margin for statements)
- MOVEF =.TRUE. ; (Move Format statement)
- RLBFMT =.TRUE. ; (Relabel Format statement)
- SEQRQD = .TRUE. ; (Add sequence numbers)
- SEQINI = 0 ; (Initial sequence number)
- SEQINC = 10 ; (Sequence number increment)

NOTES : If the user wants to execute the output of ISTPL immediatly, then it is recommended that the user does not use the option SEQEQD. The output using the SEQRQD isn't accepted by the VAX/VMS Fortran-77 compiler.

e. Elapsed CPU Time Comparison

The execution times of the different size programs are shown in TABLE II and TABLE III. To get the results, the author used almost same procedures which are listed in Appendix A.

4. ISTPT (Precision Transformer)

a. Description

ISTPT [Ref. 23: p. 3] will transform a Fortran-77 program from REAL to DOUBLE PRECISION or vice versa. The input program must have all names explicitly typed for ISTPT to work correctly. The tool ISTDS [Ref. 21: pp. 3-4] can perform this function.

TABLE II

CPU TIME COMPARISON (USING SIMPLE METHOD) OF ISTPL

(unit : secs)

program size	85 LOC	700 LOC	725 LOC
1	10.03	75.73	80.89
2	10.09	76.10	82.15
3	10.11	76.29	80.91

TABLE III

CPU TIME COMPARISON (USING COMPLEX METHOD) OF ISTPL

(unit : secs)

program size	85 LOC	700 LOC	725 LOC
1	21.03	165.87	193.28
2	20.75	167.58	198.85
3	20.90	168.43	192.85

ISTPT takes as its input the parse tree, symbol table and comment index produced by ISTYP [Ref. 18: pp. 3-14] and the comment file produced by ISTLX [Ref. 15: pp. 1-5] and produces a new token stream file. The new token and comment stream produced by ISTPT can be converted to Fortran source code using ISTPL [Ref. 20: pp. 3-18].

All warning and error message produced by ISTPT while converting the user's program are also inserted into the output token stream as comments. These begin with "C*PT*WARNING*" and *C*PT*ERROR*" respectively.

Any warning and error message produced by ISTYP while parsing the program will be displayed on the standard error channel (In the VAX/VMS system they could be displayed on the screen of the terminals. Therefore, if the user wants to keep the errors and warning messages, then use the VAX/VMS special commands. This technique will be discussed in a later section.). They will not, however ,be inserted into the output token stream as comments.

b. The Flows of The ISTPT

There are two kinds of flows to execute the ISTPT. They were named the simple method and the complex method. As mentioned above, the simple method doesn't use the ISTDS (a declaration standardizer), hence the output of using the simple method doesn't include action on the declaration parts of the given source program.

But the complex method uses the ISTDS, the output of using the complex method is more readable and more understandable than the output which result from using the simple method. The complex method takes more time than the simple method (The time comparison charts are shown by TABLE IV and TABLE V.). The user must decide on the trade offs between the quality of the solution and the execution time. The outputs of using both methods are shown by Appendix D. The flows of the ISTPT are shown by Figure 8.6 and Figure 8.7.

c. Transformation Details

This section lists the major details of the transformation performed by ISTPT. ISTPT should always produce correct output except when it detects an error, or with complex arithmetic. Although ISTPT does not attempt to ensure that the transformation will be reversible, the only difference will be where the code was originally of mixed precision, or an unusal intrinsic function (such as MAX1) was used.

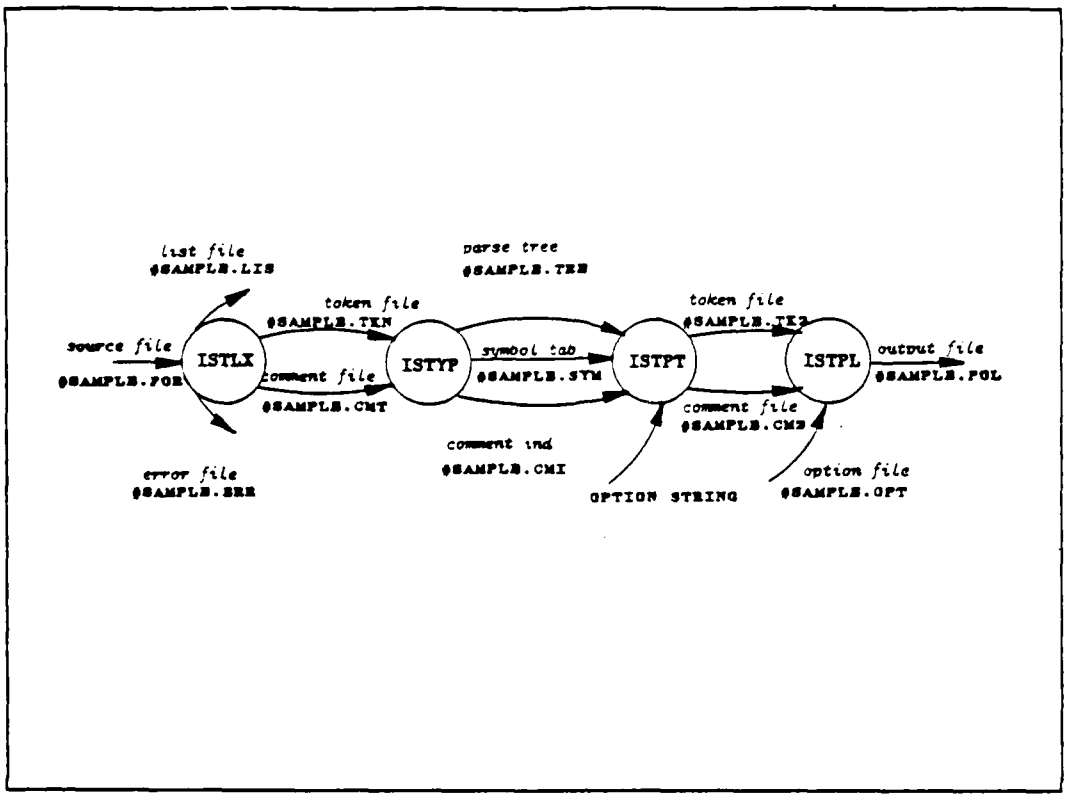


Figure 8.6 The Flows of ISTPT (Simple Method).

1) The key word REAL is changed to/from DOUBLE PRECISION

2) Real constants are transformed to/from double precision, and if appropriate "D0" will be added to or deleted from the end of the constant.

3) The E-format descriptor is transformed to/from the D-format descriptor. This will not change the "Ew.dEe" form of the E-format descriptor, as there is no D-format equivalent.

4) Complex variable usage will result in a warning message when converting to DOUBLE PRECISION (ISTPT does not attempt to transform complex expressions).

5) EQUIVALENCE statement are checked to ensure that their meaning does not change during the transformation; if it does then an error message is produced.

d. Elapsed CPU Time Comparison

The execution times of the different size programs are shown by TABLE IV and TABLE V.

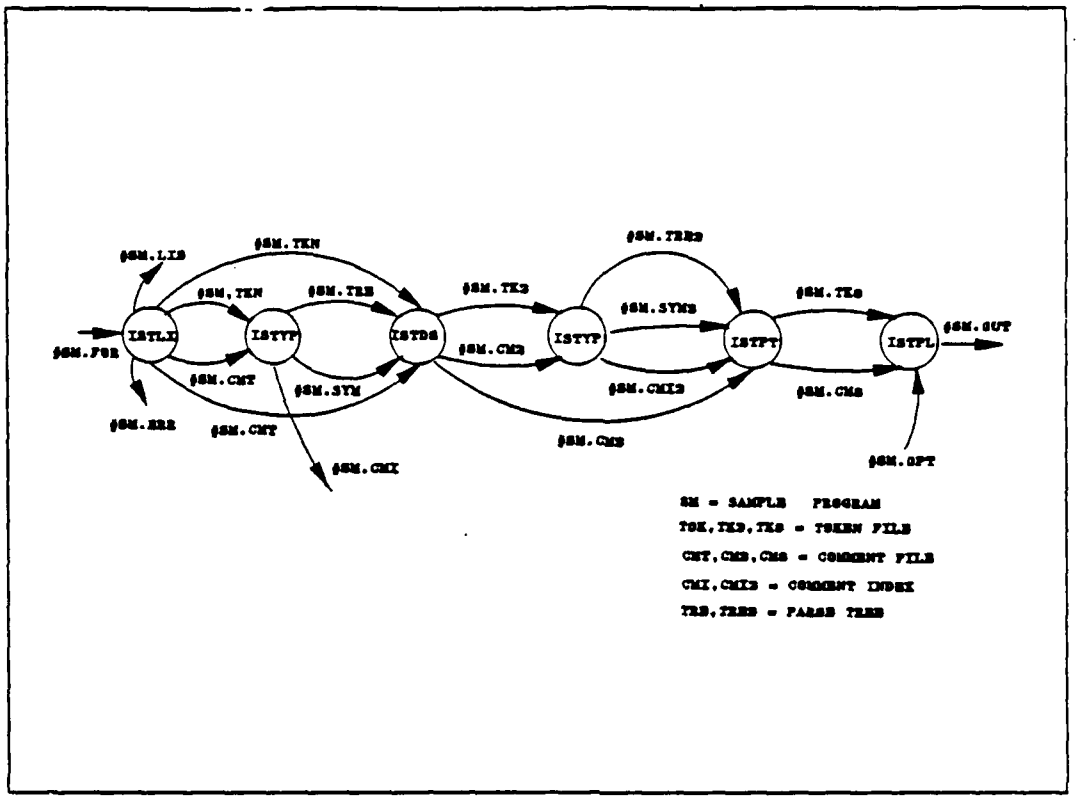


Figure 8.7 *The Flows of ISTPT (Complex Method).*

TABLE IV

CPU TIME COMPARISON (USING SIMPLE METHOD) OF ISTPT

(unit : secs)

program size	85 LOC	700 LOC	725 LOC
1	19.83	157.94	178.05
2	19.95	158.54	178.02
3	19.78	159.36	178.04

TABLE V

CPU TIME COMPARISON (USING COMPLEX METHOD) OF ISTPT

(unit : secs)

program size	85 LOC	700 LOC	725 LOC
1	30.77	252.36	305.30
2	31.14	252.59	305.28
3	31.06	254.01	305.23

5. ISTAL (Documentation Generation Aid)

a. Description

ISTAL may be used to create a number of reports on the static and dynamic analysis of a program unit or set of program units [Ref. 24: p. 3]. The reports may be automatically inserted, as requested, into a specified user document. ISTAL uses information created by other tools, notably ISTYP and ISTAN [Ref. 25: pp. 3-16].

The tool ISTAN creates static analysis information and also instruments Fortran-77 to produce dynamic information on program usage. Both the static and dynamic information created can be processed by ISTAL.

b. The Useful Commands to Operate the ISTAL

There are 20 commands which are used to get the results from ISTAL [Ref. 24: p. 9]. The commands are listed in Appendix E. The author selected some of them to demonstrate how to work the commands.

c. The Operation of ISTAL

Below these are several examples which are to demonstrate the operation of the ISTAL [Ref. 24: pp. 10-21]. But the examples of the reference is the result of the EMBEDDED regime [Ref. 13: pp. 3-4]. As mentioned above, the installation of the VAX/VMS system is the STAND-ALONE

regime. Therefore the operation procedures are a little different. Especially, the file name convention is different (see chapter 6).

The outputs of the ISTAL are displayed on the terminals. To keep the outputs of the operation, VAX/VMS commands are required. Figure 8.8 represents the commands.

```
$ SET HOST/LOG = RESULT.LIS CSVMS1
  (If you issue the command, then the system will be
   logged out. The user must log in again. After log
   in, if you work anything, then all commands and
   all outputs which are displayed on the screen
   will be recorded at the list file RESULT.LIS.)

After finishing your jobs, type the following
commands.

$ LOGOUT
  (This command returns control to the CSVMS1.)

$ PRINT RESULT.LIS
 (This command prints your jobs.)
```

Figure 8.8 VAX/VMS commands
(to obtain the outputs of ISTAL).

The ISTAL requires the symbol table; to get the symbol table, the tools ISTLX and ISTYP should be executed first. The important thing is that the ISTAL operates under the TOOLPACK command executor (ISTCE [Ref. 26: pp. 2-13]). Before invoking ISTAL, therefore, the user must invoke the ISTCE. The summarized procedures are shown by Figure 8.9.

6. ISTAN (Execution Analyzer)

```
$ISTCE          (to invoke the command executor)
ce :            (ISTCE prompt)
al 0,1          (the inputs and outputs are standard
                 i/o channel(terminal i/o))
al :            (ISTAL prompt)
FOLDING = YES
al :
VERBOSE = YES
al :
CALLGRAPH = #SAMPLE.SYM  (symbol table name)
     (The output of CALLGRAPH is Appendix E)

al :
COMMON = #SAMPLE.SYM
     (The output of COMMON is Appendix E)

al :
TABLE  = #SAMPLE.SYM
al :
SYMBOL = TEST  (TEST is the name of Main program)
     (The output of SYMBOL = TEST is Appendix E)

al :
SYMBOL = STAND (STAND is the name of the subprogram)
     (The output of SYMBOL = STAND is Appendix E)

al :
WARNING = STAND
     (The output of WARNING = STAND is Appendix E)

al :
XREFERNCE
     (The output of XREFERENCE is Appendix E)

al :
FULLXREFERENCE
     (The output of FULLXREFERECE is Appendix E)

al :
control Z       (return to command executor)
< ISTAL Normal Termination >

ce :

QT              (return to VAX/VMS system)
< TIE : Terminated >
```

Figure 8.9 ISTAL Operation Procedures.

a. Description

ISTAN [Ref. 25: p. 4] takes as its input a Fortran-77 program in token stream from (as produced by ISTLX) and produces an instrumented Fortran source, a statement summary file for input to ISTAL, an annotated token stream and a summary report.

The annotated token stream lists the segment numbers (a segment is a section of straight line code) used in later reports.

It is not necessary for the input to contain a complete Fortran program. If only a few routines are to be analyzed, they may be input to ISTAN and the instrumented output combined with the rest of the program.

b. The Flows of ISTAN

The running of the ISTAN is quite complex. The user must pay attention to the sequence of the execution procedures.

First of all, ISTAN requires the comment stream and the token stream. Therefore, the running of the ISTLX is required by ISTAN. After getting the token stream and the comment stream, execute the ISTAN, then you will get the instrumented source codes which is a Fortran-77 program and many additional lines are added to get the user's output and some additional information named SEGMENT EXECUTION FREQUENCIES (see Appendix F).

A third step is required to execute the ISTPL. The results of this step is the polished output file which is an Instrumented Fortran-77 program. Figure 8.10 shows the Data Flow Diagram of ISTAN.

c. The Useful Information of ISTAN

- The Instrumented Program : The instrumented program produces as output a listing file and an optional single-run data file. These are in addition to any output normally produced by the non-instrumented program.
- The Listing File : The file contains a formatted report of the execution frequencies of each segment in the instrumented program and a list of all segments which were not executed.
- Single-Run Data : The single-run data file is written by the instrumented program upon termination and contains the segment execute frequencies for that run in a form suitable for input to ISTAL (The single-run data file is only produced if the option is specified to ISTAN.).

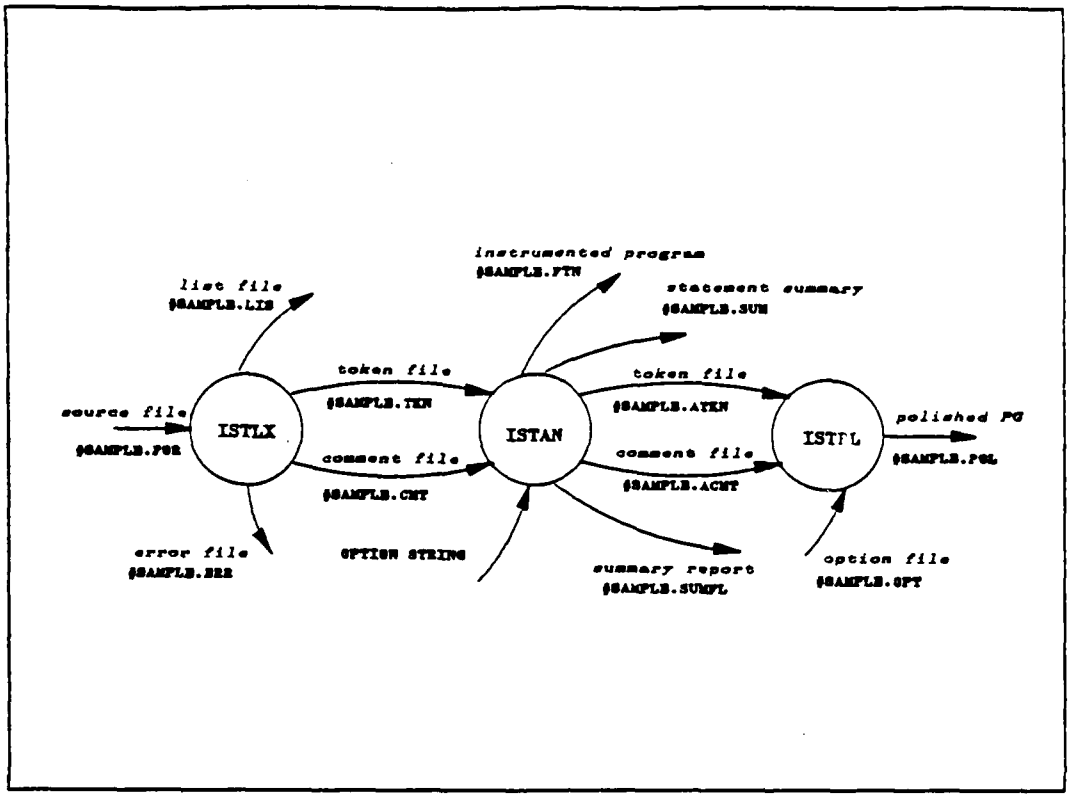


Figure 8.10 The Flows of ISTAN.

d. The Operation of ISTAN

As mentioned above, the operational procedures for ISTAL are little different with the given specification [Ref. 14: pp. 8-16]. The author selected the options, and the results are illustrated by Appendix F. The summarized operational procedures are listed in Figure 8.11 and Figure 8.12. After executing the first step correctly, the user must compile and link the instrumented source program. The following are needed :

- $ fortran SAMPLE.FTN
- $ link SAMPLE.OBJ (or SAMPLE)
- $ run SAMPLE.EXE (or SAMPLE)

The output of the above execution is shown by Appendix F. I recommend to use the ISTCE (Command Executor) for running the ISTAN.

```
$ ISTCE          (to invoke the command executor)
ce :             (ISTCE prompt)
AN               (to invoke the ISTAN)
Input token stream : #SAMPLE.TKN
Input comment stream : #SAMPLE.CMT
Output instrumented source code : #SAMPLE.FTN
Output statement summary : #SAMPLE.SUM
Output annotated token stream : #SAMPLE.ATKN
Output annotated comment stream : #SAMPLE.ACMT
Output summary file : #SAMPLE.SUMFL
Options : RUNDATA='SAMPLE'
 <ISTAN Normal Termination>
ce :
QT               (return to VAX/VMS system)
 <TIE :Terminated>
```

Figure 8.11 Operation Procedures of ISTAN (first step).

```
$ISTCE          (to invoke the command executor)
ce :            (ISTCE prompt)
PL              (to invoke the ISTPL (polish tool))
Input token stream : #SAMPLE.ATKN
      (The token stream produced by ISTAN.)

Input comment stream : #SAMPLE.ACMT
      (The comment stream produced by ISTAN.)

Polish output : #SAMPLE.APOL
Option file : (none)

    <ISTPL Normal Termination>
ce :            (return to the command executor)
AL 0,1          (to invoke the ISTAL, terminal i/o)
al :            (ISTAL prompt)
VERBOSE = YES
al :
FOLDING = YES
al :
ANNOTATED = #SAMPLE.APOL
al :
RUN = #SAMPLE.DAT
        (This file produced by the ISTAN option.)

al :
SUMMARY = #SAMPLE.SUM
al :
LISTING
    (The output of the LISTING command is Appendix F)

al :
SEGMENT = ?*
    (The output of SEGMENT = ?* is Appendix F)

al :
TOTALS = ?*
    (The output of TOTALS = ?* is Appendix F)

STATIC = TEST
    (The output of STATIC = TEST is Appendix F)

al :
DYNAMIC = TEST
    (The output of DYNAMIC = TEST is Appendix F)
al :
control Z       (return to the ISTCE)

   <ISTAL Normal Termination>
ce :
QT              (return to the VAX/VMS system)

   < TIE : Terminated >
```

Figure 8.12 Operation Procedures of ISTAN (second step).

IX. EVALUATION

A. COMPARE USER NEEDS TO TOOLPACK GOALS

The characteristics of mathematical software (definitions, application domains, need for efficiency, development environment, portability) and a description of programmers of mathematical software were presented in chapter 5. One way to compare the user needs with TOOLPACK goals is to identify any discrepancies between the characteristics of typical users of TOOLPACK and the assumptions that the designers of TOOLPACK made about them.

The TOOLPACK system assumes that users have a working knowledge of terms like "lexical analyzers","parsers", "table managers" and "report generators". As discussed in chapter 5 the typical programmer (of mathematical software) is not acquainted with these terms. The TOOLPACK documentation does not include definitions for these terms, tutorials on them or even references into the computer science literature to learn about them. The typical programmer does not have the background to easily access the literature on these topics. It will be necessary for the TOOLPACK documentation to close the gap between user needs and TOOLPACK goals by providing expository materials on all the pertinent contents.

In addition to specific concepts that are unknown to the typical user, the documentation contains phases and jargon from computer science that will be difficult for scientific programmers. It is probably not possible to remove all the technical terms from all the documentation, but the documentation could be improved greatly by identifying two distinct readers:

1. typical users
2. software experts

Each separate document could be designated for the typical user or for the expert (or perhaps installer). It would then be necessary for the user documents to systematically either remove or explain each technical term.

One specified TOOLPACK goal was to provide "a structured Fortran language which enhances standard Fortran with modern control and data structures". The need for these capabilities is overshadowed by the need for code that can be easily debugged, tested and maintained. Since the inclusion of a production quality compiler is not a realistic goal, the TOOLPACK goal could be accomplished only with a preprocessor (like for example RATFOR). As discussed above, preprocessors have serious drawbacks including the inability to relate error messages to the source code. So although users do not need a preprocessor, the goal of providing better control and data structures than available in standard Fortran is important. Some capability for an improved language are provided by the non-standard extensions to Fortran that are unique for each computer manufacturer. TOOLPACK does not have a goal to support these computer unique Fortran extensions that help provide the modern features needed by programmers.

The most serious gap between user needs and TOOLPACK goals is in portability. Although portability was only a very minor part of the original TOOLPACK goals [Ref. 1: p. 5-6] it achieved major status in the architectural design document [Ref. 7: p. 3]. Portability is only a secondary goal of the vast majority of scientific programmers; more important goals are runtime efficiency (including use of computer unique Fortran extensions) and high programmer productivity (use of simple, effective tools with low run time expense). Programmers would like to be able to move their code easily to other machines and to easily use programs produced on other computer systems but they

recognize that the main obstacle to portability is numeric precision. No goals of TOOLPACK (or any theory the author knows about) is going to solve the problem of moving high performance numerical algorithms from one computer to another computer that has a different model of real computation. It is possible to have portability for non-numeric algorithms (like TOOLPACK itself) but is a serious mistake to confuse the needs of NAG and the TOOLPACK developers for portability with the needs of TOOLPACK users.

As discussed in chapter 5, most programmers of mathematical software have a very limited domain of interest. They are interested in a limited set of software for a limited set of science, mathematics or engineering problems. They therefore need to be able to pick and choose the subset of TOOLPACK that best suits their needs. Also the suite of software tools that is provided by the computer manufacturer differs widely from machine to machine. Given a choice between a manufacturer provided tool (optimized to the machine, integrated with the compiler) and an equivalent TOOLPACK tool, most users will use the manufacturer supplied tool. Thus the TOOLPACK goals need to be subdivided into subgoals that will allow the user to pick, install and learn subsets of the total capabilities without commiting to a comprehensive environment.

B. COMPARE TOOLPACK GOALS TO ITS CAPABILITIES.

It is difficult to compare the TOOLPACK goals to its capabilities because there seems to be two very different set of goals. The original goals of TOOLPACK [Ref. 1: p. 5-6] seem to be focused on providing tools for the vast majority of mathematical software programmers that are described in chapter 5. As can be seen in chapter 4, the architectural design document [Ref. 7: p. 3] presents a different set of goals; if the phase "mathematical software" in goal 1 were changed to "non-numeric mathematical

software", the architectural design goals would be seen to be focused on an entirely different set of users namely those producing software like TOOLPACK.

If we accept the goals of the architectural design, the capabilities of TOOLPACK version 1 fulfills most of the goals. For a user population with a strong computer science background in language translation the system capabilities are well understood, the terms in the documents are familiar, the heavy use of technical jargon makes the documents appropriately terse, and the installation while needlessly complex and ambiguous is within the skills of a good systems programmer who has two weeks to spend on what should be a one day job. The lexical analyzer, parsers, table manager, pretty printer all work well and the "tool fragment" design is well suited for those users who want to build their own non-numeric mathematical software.

If we accept the original TOOLPACK goals the evaluation is very different. For the vast majority of programmers of mathematical software the capabilities of version 1 fall far short of the goals. Aside from enforcing the use of standard Fortran (the tools reject computer unique extensions and/or produce code that is incorrect), there is nothing that supports the portability of numeric software. As mentioned above, easy portability of numeric software is an unrealistic goal, however, it is striking that there is no discussion and no system capability directed toward this important problem.

The goal of a structured Fortran extension of standard Fortran was also abandoned. This was most likely a good decision; in the last several years there has been an increasing lack of enthusiasm for preprocessors. Likewise the automatic conversion of standard Fortran to structured Fortran is not a serious loss.

The capabilities for precision transformation, static and dynamic analysis tools, Fortran syntax editor, pretty printer and the language translation tools (lexical analyzers, parsers, etc) were achieved in version 1. These tools represent a real contribution to the programmer that knows how to use them.

An important and dominant capability that is not explicitly mentioned in the goals is the file handling and control environment that comes with TOOLPACK. The author used only the STAND-ALONE mode of operation for several reasons:

1. The VAX/VMS system already has a good modern file system.
2. The focus of the evaluation was on tools rather than environment.
3. The installation of even the STAND-ALONE took so much longer than planned (months instead of days) that there was not time to evaluate other modes.

Despite the lack of direct evaluation, the author feels that there is little in the other modes of operation that justify the cost and effort of learning and installing them. Given that the typical programmer wants to maximize his use of the manufacturer supplied tools and given these are good (like on the VAX/VMS), there is little incentive to use the TOOLPACK file structure or command features.

C. COMPARE USER NEEDS TO TOOLPACK CAPABILITIES.

As discussed above, the human factors associated with software products are important. Mathematical software programmers are also not eager to learn new systems with new messages and error reports. The author feels that the TOOLPACK error messages are unsatisfactory. The following is an example of TOOLPACK error messages:

```
C*PL*ERROR Unexpected Statement type
C*PL*ERROR Unexpected <TZOES>
C*PL*ERROR Internal Error (GRIND1) - TZEOS confusion
```

These messages, produced by ISTPL (the polish tool), resulted from not defining a subprogram name in the declaration part of a program. These messages don't help the user identify the error. TOOLPACK needs to develop and use a more clear error reporting system.

The TOOLPACK project consists of 33 tools. The author thinks that most mathematical programmers won't use all these tools. They will select the tools which are most convenient and helpful to them. Some tools will never be used by mathematical programmers. For example, various computer systems have their own editor system (i.e., IBM: XEDIT, VAX/VMS: EDIT, and UNIX system : VI editor). These editors are already familiar to its users and the users are skilled in their editor systems. The author, therefore, thinks that mathematical programmers will use their own editor system rather than ISTED (editing tool) which is provided by the TOOLPACK project.

The author suggests that the TOOLPACK project should concentrate on suitable tools rather than unused tools, so that the quality of tools will be more powerful than previously distributed tools. The author recommends that ISTPL, ISTPT, ISTAN, and ISTAL are very useful tools.

It is very hard to install TOOLPACK. Today most software packages are not very hard to install in their own computer systems. Generally speaking, even though the users are not specialists in the software packages, if they follow the given manual then they can easily install the package.

Sometimes, if the software packages are difficult to install then the company of the distributed program makes available installation services and/or good manuals. The author understands that the TOOLPACK project is not a commercial operation, but the installation guides (for example: TIE code installer's guide and Tool installer's guide) are not very easy to read and understand. Another

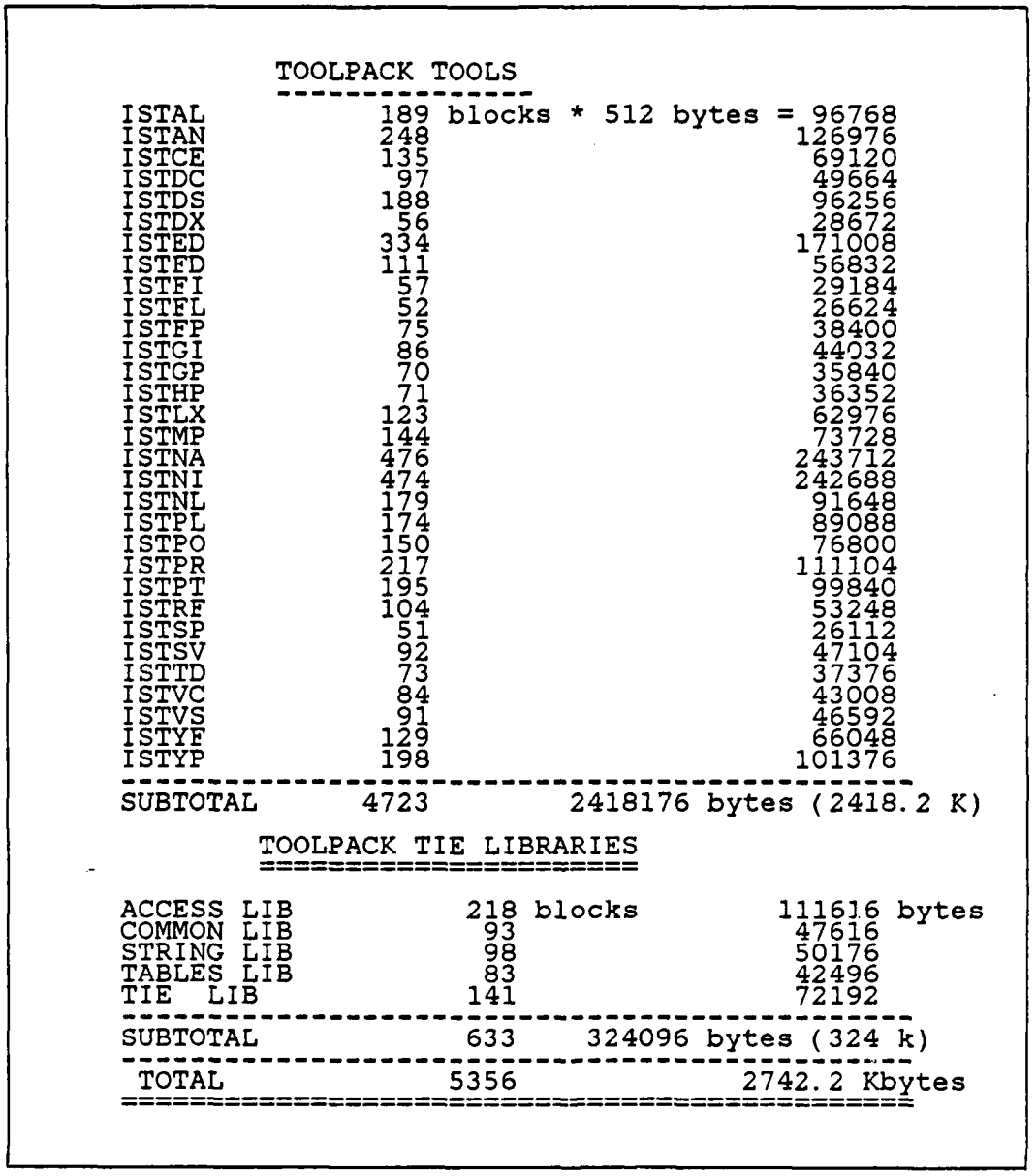
thing is that to understand the TOOLPACK installer's guide the reader needs a background of computer architecture and operating systems. The author spent 4 months to install the TOOLPACK with help from our computer center staff. There is an immediate need for a tool to help TOOLPACK users install and test the system.

D. COSTS VERSUS BENEFITS

The system resources needed by TOOLPACK version 1 are considerable. The memory requirements are significant, this would have major impact for small computer systems. The details of memory requirements are shown in Figure 9.1.

The cost and effort involved in initially constructing the tools are significant. Any modification of the programs involves significant effort and modifications in the library routines involves significant work. These jobs are all tedious and time consuming and are well beyond the patience of a single mathematical software programmer. They can only be justified if many people are using the system. The potential user population is reduced significantly if use is restricted to installations with a systems programmer who is responsible for installation and maintenance.

The benchmarks in chapter 8 show that CPU times are very large for even medium size programs. This will surely drive users to using software provided by the computer manufacturer whenever possible. In particular, the users are likely to use file handling and control features that are provided more cheaply on their computer.



TOOLPACK TOOLS

ISTAL	189	blocks * 512 bytes =	96768
ISTAN	248		126976
ISTCE	135		69120
ISTDC	97		49664
ISTDS	188		96256
ISTDX	56		28672
ISTED	334		171008
ISTFD	111		56832
ISTFI	57		29184
ISTFL	52		26624
ISTFP	75		38400
ISTGI	86		44032
ISTGP	70		35840
ISTHP	71		36352
ISTLX	123		62976
ISTMP	144		73728
ISTNA	476		243712
ISTNI	474		242688
ISTNL	179		91648
ISTPL	174		89088
ISTPO	150		76800
ISTPR	217		111104
ISTPT	195		99840
ISTRF	104		53248
ISTSP	51		26112
ISTSV	92		47104
ISTTD	73		37376
ISTVC	84		43008
ISTVS	91		46592
ISTYF	129		66048
ISTYP	198		101376
SUBTOTAL	4723	2418176 bytes	(2418.2 K)

TOOLPACK TIE LIBRARIES

ACCESS LIB	218	blocks	111616 bytes
COMMON LIB	93		47616
STRING LIB	98		50176
TABLES LIB	83		42496
TIE LIB	141		72192
SUBTOTAL	633	324096 bytes	(324 k)
TOTAL	5356		2742.2 Kbytes

Figure 9.1 Memory spaces of each tool.

X. CONCLUSIONS

One notable lack in the TOOLPACK project is a comprehension user profile that would describe the user's needs, capabilities and potential for learning new ideas. A well known tenet in software engineering is that in the absence of good user requirements, designers and implementers use themselves as a model of the user. On several issues, most notably portability, the TOOLPACK goals and capabilities are heavily skewed towards users who are producing non-numeric software to be distributed (NAG and the "PACK" developers) and away from users who are writing numeric software for their own use.

The vast majority of mathematical software programmers are producing software for their personal use. They have a strong preference for using software tools that are provided by the manufacturer of their computer because these tools are very efficient and are (and will continue to be) integrated with the Fortran compiler and its computer unique extensions.

For these users, TOOLPACK provides capabilities that are not currently available on their computer; they are more interested in picking and choosing specific tools than in learning (another) total environment. Another (but different) set of users is interested in producing portable numeric software for general use, in the author's opinion they are better served by having tools that help them customize software to different environments (i.e., machines, compilers) rather than build software for a standard environment that is not likely to be widely accepted by the vast majority of mathematical software programmers.

A third (and very small) set of users is interested in producing non-numeric software (e.g., TOOLPACK) for general use.

For these users (unlike the others) portability is both technically possible and economically desirable. While waiting for their computer manufacturer to produce new tool X, these users are quite willing to pay the run time cost of using portable tools if they are not forced to pay a heavy cost to install and are not forced to use a new comprehensive environment.

In conclusion, the capabilities of TOOLPACK release 1 do not fulfill the original goals of supporting scientific programmers in the construction of numerical software. The system is much too hard to install, demands the user accept a new environment and does not make effective use of computer resources. The fundamental problem with the system is that it has been driven entirely by the goal of producing a portable environment- this is not a major need of scientific programmers. It is difficult to see how modifications of the present system can lead to a system that will meet user needs, it appears that a major rethinking of goals is required.

APPENDIX A

VAX/VMS COMMAND FILE (EXAMPLE FOR BENCHMARK TEST)

```
$!
$!  ISTLX.COM :   command file to execute ISTLX
$!
$TIE$ERROR = " "
$!
$ IF P1.EQS." " THEN INQUIRE P1 "NAME OF THE FORTRAN FILE"
$!
$SHOW SYSTEM
$!
$!
$ ISTLX  #'P1'.FOR  #'P1'.LIS  #'P1'.ERR  #'P1'.TKN
$        #'P1'.CMT
$ IF TIE$ERROR .EQS. "ERROR" THEN EXIT
$!
$SHOW SYSTEM
```

APPENDIX B

USEFUL COMMAND FILES

1. ISTDC

```
$!
$! DC.COM : Command file to execute the ISTDC
$!
$! expects one parameter, the file name of the Fortran
$! file with no extension
$!
$! PREREQUISITES : NONE
$! NOTE          : The name of the output file is DIFF.LIS
$!
$ TIE$ERROR=""
$!
$ IF P1 .EQS. "" THEN INQUIRE P1 "Name of Standard File"
$ IF P2 .EQS. "" THEN INQUIRE P2 "Name of Comparison File"
$!
$ ISTDC #'P1'.FOR  #'P2'.FOR  #DIFF.LIS
$ IF TIE$ERROR .EQS. "ERROR" THEN EXIT
```

2. ISTFD

```
$!
$! FD.COM : Command file to execute the ISTFD
$!
$! expects one parameter, the file name of the Fortran
$! file with no extension
$!
$! PREREQUISITES : NONE
$!
$ TIE$ERROR=""
$!
$ IF P1 .EQS. "" THEN INQUIRE P1 "Name of First Token
                                  File"
$ IF P2 .EQS. "" THEN INQUIRE P2 "Name of Second Token
                                  File"
$!
$ ISTFD #'P1'.TKN  #'P1'.CMT  #'P2'.TKN  #'P2'.CMT
        #'P1'.LIS
$  IF TIE$ERROR .EQS. "ERROR" THEN EXIT
```

3. ISTFP

```
$!
$! FP.COM : Command file to execute the ISTFP
$!
$! This tool is the fast polish program (Within the
$!  GENERAL tools).
$!
$! PREREQUISITES : ISTLX
$!
$ TIE$ERROR=""
$!
$ IF P1 .EQS. "" THEN INQUIRE P1 "Name of Fortran File"
$!
$!
$ ISTLX  #'P1'.FOR  #'P1'.LIS #'P1'.ERR  #'P1'.TKN
$        #'P1'.CMT
$ IF TIE$ERROR .EQS. "ERROR" THEN EXIT
$!
$ ISTFP #'P1'.TKN  #'P1'.CMT  #'P1'.OUT
$ IF TIE$ERROR .EQS. "ERROR" THEN EXIT
```

4. ISTLX

```
$!
$! LX.COM : Command file to execute the ISTLX
$!
$! Expect one parameter, the file name of the Fortran file
$!  with no extension
$!
$! PREREQUISITES : NONE
$!
$ TIE$ERROR=""
$!
$ IF P1 .EQS. "" THEN INQUIRE P1 "Name of Fortran File"
$!
$!
$ ISTLX  #'P1'.FOR  #'P1'.LIS #'P1'.ERR  #'P1'.TKN
$        #'P1'.CMT
$ IF TIE$ERROR .EQS. "ERROR" THEN EXIT
$!
```

5. ISTPL (Simple Method)

```
$!
$!  PL1.COM : Command file to polish a single file of
$!            Fortran-77 code
$!
$!  PREREQUISITES : ISTLX
$!
$!  NOTE : This command file doesn't use the tools which
$!         are ISTYP and ISTDS. Therefore, the output is
$!         less readable than the other output which uses
$!         the ISTYP and ISTDS. If the user wants to use the
$!         option file then this command file needs
$!         minor correction which is to append the option
$!         file, that is get from the tool ISTPO.
$!
$ TIE$ERROR=""
$!
$ IF P1 .EQS. "" THEN INQUIRE P1 "Name of Fortran File"
$!
$ ISTLX  #'P1'.FOR  #'P1'.LIS #'P1'.ERR  #'P1'.TKN
$        #'P1'.CMT
$ IF TIE$ERROR .EQS. "ERROR" THEN EXIT
$!
$ ISTPL  #'P1'.TKN  #'P1'.CMT #'P1'.POL
$ IF TIE$ERROR .EQS. "ERROR" THEN EXIT
```

6. ISTPL (Complex Method)

```
$!
$!  PL2.COM : Command file to polish a single file of
$!            Fortran-77 code
$!
$!  PREREQUISITES : ISTLX,ISTYP, and ISTDS
$!
$!  NOTE : This command file  uses the tools which are
$!         ISTYP and ISTDS. Therefore, the output is more
$!         readable than the other output which doesn't use
$!         the ISTYP and ISTDS. If the user wants to use the
$!         option file then this command file needs a
$!         minor correction which is to append the option
$!         file, that is get from the tool ISTPO.
$!
$ TIE$ERROR=""
$!
$ IF P1 .EQS. "" THEN INQUIRE P1 "Name of Fortran File"
$!
$ ISTLX  #'P1'.FOR  #'P1'.LIS #'P1'.ERR  #'P1'.TKN
$        #'P1'.CMT
$ IF TIE$ERROR .EQS. "ERROR" THEN EXIT
$!
$ ISTYP  #'P1'.TKN  #'P1'.CMT #'P1'.TRE  #'P1'.SYM
$        #'P1'.CMI
$ IF TIE$ERROR .EQS. "ERROR" THEN EXIT
$!
$ ISTDS  #'P1'.TRE  #'P1'.SYM #'P1'.TKN  #'P1'.CMT
$        #'P1'.TK2  #'P1'.CM2
$ IF TIE$ERROR .EQS. "ERROR" THEN EXIT
$!
$ ISTPL  #'P1'.TKN  #'P1'.CMT #'P1'.POL
$ IF TIE$ERROR .EQS. "ERROR" THEN EXIT
```

7. ISTPT (Simple Method)

```
$!
$! PT1.COM : Command file to Precision-transform from a
$!           single precision file of Fortran-77 source
$!           to double precision file of Fortran-77
$!           source code.
$!
$! PREREQUISITES : ISTLX (lexical analyzer)
$!                  ISTYP (parser)
$!                  ISTPT (precision transformer)
$!                  ISTPL (polish)
$!                        ((default options are none))
$!
$! NOTE : This command file  doesn't use the tool which
$!        is the ISTDS. Therefore, the output is less
$!        readable than the other output which uses the
$!        ISTDS.
$!
$ TIE$ERROR=""
$!
$ IF P1 .EQS. "" THEN INQUIRE P1 "Name of Fortran File"
$!
$!
$ ISTLX  #'P1'.FOR  #'P1'.LIS #'P1'.ERR  #'P1'.TKN
$        #'P1'.CMT
$ IF TIE$ERROR .EQS. "ERROR" THEN EXIT
$!
$ ISTYP  #'P1'.TKN  #'P1'.CMT #'P1'.TRE  #'P1'.SYM
$        #'P1'.CMI
$ IF TIE$ERROR .EQS. "ERROR" THEN EXIT
$!
$ ISTPT  #'P1'.TRE  #'P1'.SYM #'P1'.CMI  #'P1'.CMT
$        #'P1'.TK2  #'P1'.CM2
$ IF TIE$ERROR .EQS. "ERROR" THEN EXIT
$!
$ ISTPL  #'P1'.TKN  #'P1'.CMT #'P1'.POL
$ IF TIE$ERROR .EQS. "ERROR" THEN EXIT
```

8. ISTPT (Complex Method)

```
$!
$! PT2.COM : Command file to Precision-transform from a
$!           single precision file of Fortran-77 source
$!           to double precision file of Fortran-77
$!           source code.
$!
$! PREREQUISITES : ISTLX (lexical anayzer)
$!                  ISTYP (parser)
$!                  ISTDS (declaration standardizer)
$!                  ISTPT (precision transformer)
$!                  ISTPL (polish)
$!                        ((default options are none))
$!
$! NOTE : This command file  uses the tool which is the
$!        ISTDS. Therefore, the output is more readable
$!        than the other output which doesn't use the ISTDS
$!
$ TIE$ERROR=""
$!
$ IF P1 .EQS. "" THEN INQUIRE P1 "Name of Fortran File"
$!
$!
$ ISTLX  #'P1'.FOR  #'P1'.LIS #'P1'.ERR  #'P1'.TKN
$        #'P1'.CMT
$ IF TIE$ERROR .EQS. "ERROR" THEN EXIT
$!
$ ISTYP  #'P1'.TKN  #'P1'.CMT #'P1'.TRE  #'P1'.SYM
$        #'P1'.CMI
$ IF TIE$ERROR .EQS. "ERROR" THEN EXIT
$!
$ ISTDS  #'P1'.TRE  #'P1'.SYM #'P1'.TKN  #'P1'.CMT
$        #'P1'.TK2  #'P1'.CM2
$ IF TIE$ERROR .EQS. "ERROR" THEN EXIT
$!
$ ISTYP  #'P1'.TK2  #'P1'.CM2 #'P1'.TR2  #'P1'.SM2
$        #'P1'.CMI2
$ IF TIE$ERROR .EQS. "ERROR" THEN EXIT
$!
$ ISTPT  #'P1'.TR2  #'P1'.SM2 #'P1'.CMI2  #'P1'.CM2
$        #'P1'.TK3  #'P1'.CM3
$ IF TIE$ERROR .EQS. "ERROR" THEN EXIT
$!
$ ISTPL  #'P1'.TK3  #'P1'.CM3 #'P1'.POL
$ IF TIE$ERROR .EQS. "ERROR" THEN EXIT
```

9. ISTTD

```
$!
$! TD.COM : Text Differencer
$!          (This is the standard text differencing tool)
$!
$! expects one parameter, the file name of the Fortran
$! file with no extension
$!
$! PREREQUISITES : NONE
$!
$ TIE$ERROR=""
$!
$ IF P1 .EQS. "" THEN INQUIRE P1 "First Input File"
$ IF P2 .EQS. "" THEN INQUIRE P2 "Second Input File"
$!
$ ISTTD #'P1'.FOR  #'P2'.FOR
$ IF TIE$ERROR .EQS. "ERROR" THEN EXIT
```

10. ISTVS

```
$!
$! VS.COM : Command file to execute the ISTVS
$!
$! expects one parameter, the file name of the Fortran
$! file with no extension
$!
$! PREREQUISITES : ISTLX, ISTYP
$!
$ TIE$ERROR=""
$!
$ IF P1 .EQS. "" THEN INQUIRE P1 "Name of Fortran File"
$!
$ ISTLX  #'P1'.FOR  #'P1'.LIS #'P1'.ERR  #'P1'.TKN
$        #'P1'.CMT
$ IF TIE$ERROR .EQS. "ERROR" THEN EXIT
$!
$ ISTYP  #'P1'.TKN  #'P1'.CMT #'P1'.TRE  #'P1'.SYM
$        #'P1'.CMI
$ IF TIE$ERROR .EQS. "ERROR" THEN EXIT
$!
$ ISTVS  #'P1'.SYM  #'P1'.LIS HEAD
$ IF TIE$ERROR .EQS. "ERROR" THEN EXIT
```

11. ISTYF

```
$!
$! YF.COM : Command file to execute the ISTYF
$!
$! expects one parameter, the file name of the Fortran
$! file with no extension
$!
$! PREREQUISITES : ISTLX, ISTYP
$!
$ TIE$ERROR=""
$!
$ IF P1 .EQS. "" THEN INQUIRE P1 "Name of Fortran File"
$!
$ ISTLX  #'P1'.FOR  #'P1'.LIS #'P1'.ERR  #'P1'.TKN
$        #'P1'.CMT
$ IF TIE$ERROR .EQS. "ERROR" THEN EXIT
$!
$ ISTYP  #'P1'.TKN  #'P1'.CMT #'P1'.TRE  #'P1'.SYM
$        #'P1'.CMI
$ IF TIE$ERROR .EQS. "ERROR" THEN EXIT
$!
$ ISTYF  #'P1'.TRE  #'P1'.SYM #'P1'.CMI  #'P1'.CMT  #YF.TKN
$!       #YF.CMT
$ IF TIE$ERROR .EQS. "ERROR" THEN EXIT
```

12. ISTYP

```
$!
$! YP.COM : Command file to execute the ISTYP (Parser)
$!
$! expects one parameter, the file name of the Fortran
$! file with no extension
$!
$! PREREQUISITES : ISTLX
$!
$ TIE$ERROR=""
$!
$ IF P1 .EQS. "" THEN INQUIRE P1 "Name of Fortran File"
$!
$ ISTLX  #'P1'.FOR  #'P1'.LIS #'P1'.ERR  #'P1'.TKN
$        #'P1'.CMT
$ IF TIE$ERROR .EQS. "ERROR" THEN EXIT
$!
$ ISTYP  #'P1'.TKN  #'P1'.CMT #'P1'.TRE  #'P1'.SYM
$        #'P1'.CMI
$ IF TIE$ERROR .EQS. "ERROR" THEN EXIT
```

APPENDIX C

THE RESULTS OF RUNNING IN ISTPL

1. The Source Program for Testing

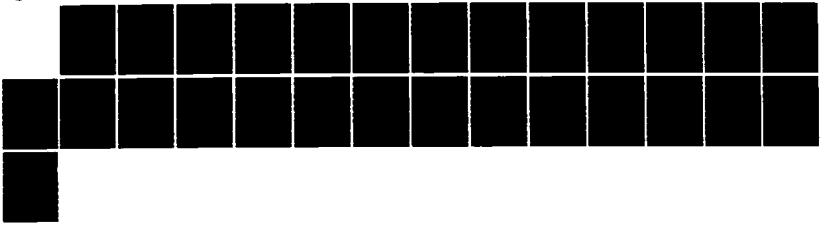
```
      PROGRAM TEST
      INTEGER X(10), NEVEN, ND3
      REAL SUM, RESULT
      WRITE(*,*) 'Please input ten integers :'
      READ(*,*) X
      RESULT = SUM(X,10)
      WRITE(*,100) RESULT
100   FORMAT(2X, 'The sum is :', F7.2)
      RESULT = NEVEN(X,10)
      WRITE(*, 150) RESULT
150   FORMAT(2X, F7.2, '   of them were even.')
      RESULT = ND3(X,10)
      IF (RESULT.EQ.0) THEN
      WRITE(*,160)
160   FORMAT(2X, 'None were divisible by 3.')
      ELSE IF (RESULT.EQ.1) THEN
      WRITE(*,170)
170   FORMAT(2X, 'One was divisible by 3.')
      ELSE
         WRITE(*,180) RESULT
180   FORMAT(2X, F7.2, '  were divisible by 3.')
      END IF
      STOP
      END


      REAL FUNCTION SUM(A,N)
      INTEGER N,A(N),I
      SUM = 0
      DO 100 I = 1, N
100      SUM = SUM + A(I)
      END

      INTEGER FUNCTION NEVEN(A,N)
      INTEGER N, A(N),I
      NEVEN = 0
      DO 100 I = 1,N
100      IF (MOD(A(I),2).EQ.0) NEVEN = NEVEN + 1
      END

      INTEGER FUNCTION ND3(A,N)
      INTEGER N,A(N),I
      ND3 = 0
      DO 100 I = 1, N
100      IF (MOD(A(I),3).EQ.0) ND3 = ND3 + 1
      END
```

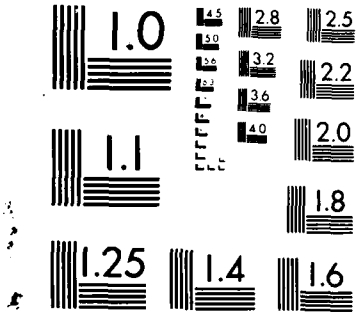
2. The Output of Polished Program(Simple Method)

```
      PROGRAM TEST
      INTEGER X(10), NEVEN, ND3
      REAL SUM, RESULT

      WRITE (*,*) 'Please input ten integers :'
      READ (*,*) X
      RESULT = SUM(X,10)
      WRITE (*,100) RESULT

100   FORMAT (2X, 'The sum is :', F7.2)
      RESULT = NEVEN(X,10)
      WRITE (*, 150) RESULT

150   FORMAT (2X, F7.2, '  of them were even.')
      RESULT = ND3(X,10)
      IF (RESULT.EQ.0) THEN
          WRITE (*,160)

160       FORMAT (2X, 'None were divisible by 3.')
      ELSE IF (RESULT.EQ.1) THEN
      WRITE (*,170)

170   FORMAT (2X, 'One was divisible by 3.')

      ELSE
          WRITE (*,180) RESULT

180   FORMAT(2X, F7.2,'  were divisible by 3.')
      END IF

      STOP

      END


      REAL FUNCTION SUM(A,N)
      INTEGER N,A(N),I

      SUM = 0
      DO 100 I = 1, N
100   SUM = SUM + A(I)
      END

      INTEGER FUNCTION NEVEN(A,N)
      INTEGER N, A(N),I

      NEVEN = 0
      DO 100 I = 1,N
100   IF (MOD(A(I),2).EQ.0) NEVEN = NEVEN + 1
      END

      INTEGER FUNCTION ND3(A,N)
      INTEGER N,A(N),I

      ND3 = 0
      DO 100 I = 1, N
100   IF (MOD(A(I),3).EQ.0) ND3 = ND3 + 1
      END
```

AD-A173 943 EVALUATION OF THE TOOLPACK FORTRAN PROGRAMMING ENVIRONMENT(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA J S KIM JUN 86 2/2

UNCLASSIFIED F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

3. The Output of Polished Program(Complex Method)

```
      PROGRAM TEST
C     .. Local Scalars ..
      REAL  RESULT
C     ..
C     .. Local Arrays ..
      INTEGER X(10)
C     ..
C     ..External Functions ..
      REAL SUM
      INTEGER ND3, NEVEN
      EXTERNAL SUM,ND3, NEVEN
C     ..
      WRITE (*,*) 'Please input ten integers :'
      READ (*,*) X
      RESULT = SUM(X,10)
      WRITE (*,100) RESULT

 100  FORMAT (2X, 'The sum is :', F7.2)
      RESULT = NEVEN(X,10)
      WRITE (*, 150) RESULT

 150  FORMAT (2X, F7.2, '  of them were even.')
      RESULT = ND3(X,10)
      IF (RESULT.EQ.0) THEN
          WRITE (*,160)

 160      FORMAT (2X, 'None were divisible by 3.')
      ELSE IF (RESULT.EQ.1) THEN
      WRITE (*,170)

 170  FORMAT (2X, 'One was divisible by 3.')

      ELSE
         WRITE (*,180) RESULT

 180  FORMAT(2X, F7.2,'  were divisible by 3.')
      END IF

      STOP

      END


      REAL FUNCTION SUM(A,N)
C     .. Scalar Arguments ..
      INTEGER N
C     ..
C     .. Array Arguments ..
      INTEGER A(N)
C     ..
C     .. Local Scalars ..
      INTEGER I
C     ..
      SUM = 0
      DO 100 I = 1, N
 100  SUM = SUM + A(I)
      END

      INTEGER FUNCTION NEVEN(A,N)
C     .. Scalar Arguments ..
      INTEGER N
C     ..
C     .. Array Arguments ..
      INTEGER  A(N)
C     ..
C     .. Local Scalars ..
      INTEGER I
C     ..
```

```
C      .. Intrinsic Functions ..
       INTRINSIC MOD
C      ..
       NEVEN = 0
       DO 100 I = 1,N
  100  IF (MOD(A(I),2).EQ.0) NEVEN = NEVEN + 1
       END

       INTEGER FUNCTION ND3(A,N)
C      .. Scalar Arguments ..
       INTEGER N
C      ..
C      .. Array Arguments ..
       INTEGER A(N)
C      ..
C      .. Local Scalars ..
       INTEGER I
C      ..
C      .. Intrinsic Functions ..
       INTRINSIC MOD
       ND3 = 0
       DO 100 I = 1, N
  100  IF (MOD(A(I),3).EQ.0) ND3 = ND3 + 1
       END
```

4. The Output of Polished Program(With Errors)

```
      PROGRAM TEST
C     .. Local Scalars ..
      REAL  RESULT
C     ..
C     .. Local Arrays ..
      INTEGER X(10)
C     ..
C     .. External Functions ..
C*PL*ERROR* Unexpected statement type
C*PL*ERROR* Unexpected <TZEOS>
      SUMINTEGER ND3, NEVEN
C*PL*ERROR* Internal Error (GRIND1) - TZEOS confusion
      EXTERNAL SUM,ND3,NEVEN
C     ..
      WRITE (*,*) 'Please input ten integers :'
      READ (*,*) X
      RESULT = SUM(X,10)
      WRITE (*,100) RESULT

 100  FORMAT (2X, 'The sum is :', F7.2)
      RESULT = NEVEN(X,10)
      WRITE (*, 150) RESULT

 150  FORMAT (2X, F7.2, '  of them were even.')
      RESULT = ND3(X,10)
      IF (RESULT.EQ.0) THEN
          WRITE (*,160)

 160      FORMAT (2X, 'None were divisible by 3.')
      ELSE IF (RESULT.EQ.1) THEN
      WRITE (*,170)

 170  FORMAT (2X, 'One was divisible by 3.')

      ELSE
         WRITE (*,180) RESULT

 180  FORMAT(2X, F7.2,'  were divisible by 3.')
      END IF

      STOP

      END


      REAL FUNCTION SUM(A,N)
C     .. Scalar Arguments ..
      INTEGER N
C     ..
C     .. Array Arguments ..
      INTEGER A(N)
C     ..
C     .. Local Scalars ..
      INTEGER I
C     ..
      SUM = 0
         DO 100 I = 1, N
 100  SUM = SUM + A(I)
      END

      INTEGER FUNCTION NEVEN(A,N)
C     .. Scalar Arguments ..
      INTEGER N
C     ..
C     .. Array Arguments ..
      INTEGER  A(N)
C     ..
C     .. Local Scalars ..
```

```
      INTEGER I
C     ..
C     .. Intrinsic Functions ..
      INTRINSIC MOD
C     ..
      NEVEN = 0
      DO 100 I = 1,N
  100 IF (MOD(A(I),2).EQ.0) NEVEN = NEVEN + 1
      END

      INTEGER FUNCTION ND3(A,N)
C     .. Scalar Arguments ..
      INTEGER N
C     ..
C     .. Array Arguments ..
      INTEGER A(N)
C     ..
C     .. Local Scalars ..
      INTEGER I
C     ..
C     .. Intrinsic Functions ..
      INTRINSIC MOD
      ND3 = 0
      DO 100 I = 1, N
  100 IF (MOD(A(I),3).EQ.0) ND3 = ND3 + 1
      END
```

5. The Output of Polished Program

(With Selected Options)

```
      PROGRAM TEST                                                  TEST    0
C     .. Local Scalars ..                                           TEST   10
      REAL  RESULT                                                  TEST   20
C     ..                                                            TEST   30
C     .. Local Arrays ..                                            TEST   40
      INTEGER X(10)                                                 TEST   50
C     ..                                                            TEST   60
C     ..External Functions ..                                       TEST   70
      REAL SUM                                                      TEST   80
      INTEGER ND3, NEVEN                                            TEST   90
      EXTERNAL SUM,ND3, NEVEN                                       TEST  100
C     ..                                                            TEST  110
      WRITE (*,*) 'Please input ten integers :'                     TEST  120
      READ (*,*) X                                                  TEST  130
      RESULT = SUM(X,10)                                            TEST  140
      WRITE (*,9000) RESULT                                         TEST  150
      RESULT = NEVEN(X,10)                                          TEST  160
      WRITE (*, 9010) RESULT                                        TEST  170
      RESULT = ND3(X,10)                                            TEST  180
      IF (RESULT.EQ.0) THEN                                         TEST  190
                WRITE (*,9020)                                      TEST  200
                                                                    TEST  210
      ELSE IF (RESULT.EQ.1) THEN                                    TEST  220
                WRITE (*,9030)                                      TEST  230
                                                                    TEST  240
      ELSE                                                          TEST  250
                WRITE (*,9040) RESULT                               TEST  260
      END IF                                                        TEST  270
                                                                    TEST  280
      STOP                                                          TEST  290
                                                                    TEST  300
 9000 FORMAT (2X, 'The sum is :', F7.2)                             TEST  310
 9010 FORMAT (2X, F7.2, '  of them were even.')                     TEST  320
 9020 FORMAT (2X, 'None were divisible by 3.')                      TEST  330
 9030 FORMAT (2X, 'One was divisible by 3.')                        TEST  340
 9040 FORMAT(2X, F7.2,'  were divisible by 3.')                     TEST  350
      END                                                           TEST  360
                                                                    SUM     0
                                                                    SUM    10
      REAL FUNCTION SUM(A,N)                                        SUM    20
C     .. Scalar Arguments ..                                        SUM    30
      INTEGER N                                                     SUM    40
C     ..                                                            SUM    50
C     .. Array Arguments ..                                         SUM    60
      INTEGER A(N)                                                  SUM    70
C     ..                                                            SUM    80
C     .. Local Scalars ..                                           SUM    90
                                                                    SUM   100
      INTEGER I                                                     SUM   110
C     ..                                                            SUM   120
      SUM = 0                                                       SUM   130
      DO 100 I = 1, N                                               SUM   140
  100 SUM = SUM + A(I)                                              SUM   150
      END                                                           SUM   160
                                                                    NEVE    0
      INTEGER FUNCTION NEVEN(A,N)                                   NEVE   10
C     .. Scalar Arguments ..                                        NEVE   20
      INTEGER N                                                     NEVE   30
C     ..                                                            NEVE   40
C     .. Array Arguments ..                                         NEVE   50
      INTEGER  A(N)                                                 NEVE   60
C     ..                                                            NEVE   70
C     .. Local Scalars ..                                           NEVE   80
      INTEGER I                                                     NEVE   90
C     ..                                                            NEVE  100
C     .. Intrinsic Functions ..                                     NEVE  110
```

```
      INTRINSIC MOD                                                     NEVE  120
C     ..                                                                NEVE  130
      NEVEN = 0                                                         NEVE  140
      DO 100 I = 1,N                                                    NEVE  150
  100 IF (MOD(A(I),2).EQ.0) NEVEN = NEVEN + 1                           NEVE  160
      END                                                               NEVE  170
                                                                        ND3     0
      INTEGER FUNCTION ND3(A,N)                                         ND3    10
C     .. Scalar Arguments ..                                            ND3    20
      INTEGER N                                                         ND3    30
C     ..                                                                ND3    40
C     .. Array Arguments ..                                             ND3    50
      INTEGER A(N)                                                      ND3    60
C     ..                                                                ND3    70
C     .. Local Scalars ..                                               ND3    80
      INTEGER I                                                         ND3    90
C     ..                                                                ND3   100
C     .. Intrinsic Functions ..                                         ND3   110
      INTRINSIC MOD                                                     ND3   120
      ND3 = 0                                                           ND3   130
      DO 100 I = 1, N                                                   ND3   140
  100 IF (MOD(A(I),3).EQ.0) ND3 = ND3 + 1                               ND3   150
      END                                                               ND3   160
```

APPENDIX D

THE RESULTS OF RUNNING IN ISTPT

1. The Output of Precision Transformation

(Simple Method)

```
      PROGRAM TEST
      INTEGER X(10), NEVEN, ND3
      DOUBLE PRECISION SUM, RESULT

      WRITE (*,*) 'Please input ten integers :'
      READ (*,*) X
      RESULT = SUM(X,10)
      WRITE (*,100) RESULT

100   FORMAT (2X, 'The sum is :', F7.2)
      RESULT = NEVEN(X,10)
      WRITE (*, 150) RESULT

150   FORMAT (2X, F7.2, '  of them were even.')
      RESULT = ND3(X,10)
      IF (RESULT.EQ.0) THEN
          WRITE (*,160)

160       FORMAT (2X, 'None were divisible by 3.')
      ELSE IF (RESULT.EQ.1) THEN
      WRITE (*,170)

170   FORMAT (2X, 'One was divisible by 3.')

      ELSE
          WRITE (*,180) RESULT

180   FORMAT(2X, F7.2,'  were divisible by 3.')
      END IF

      STOP

      END
```

```
      DOUBLE PRECISION  FUNCTION SUM(A,N)
      INTEGER N,A(N),I

      SUM = 0
      DO 100 I = 1, N
100   SUM = SUM + A(I)
      END

      INTEGER FUNCTION NEVEN(A,N)
      INTEGER N, A(N),I

      NEVEN = 0
      DO 100 I = 1,N
100   IF (MOD(A(I),2).EQ.0) NEVEN = NEVEN + 1
      END

      INTEGER FUNCTION ND3(A,N)
      INTEGER N,A(N),I

      ND3 = 0
      DO 100 I = 1, N
100   IF (MOD(A(I),3).EQ.0) ND3 = ND3 + 1
      END
```

2. The Output of Precision Transformation

(Complex Method)

```
      PROGRAM TEST
C     .. Local Scalars ..
      DOUBLE PRECISION  RESULT
C     ..
C     .. Local Arrays ..
      INTEGER X(10)
C     ..
C     ..External Functions ..
      DOUBLE PREIESION  SUM
      INTEGER ND3, NEVEN
      EXTERNAL SUM,ND3, NEVEN
C     ..
      WRITE (*,*) 'Please input ten integers :'
      READ (*,*) X
      RESULT = SUM(X,10)
      WRITE (*,100) RESULT

 100  FORMAT (2X, 'The sum is :', F7.2)
      RESULT = NEVEN(X,10)
      WRITE (*, 150) RESULT

 150  FORMAT (2X, F7.2, '  of them were even.')
      RESULT = ND3(X,10)
      IF (RESULT.EQ.0) THEN
          WRITE (*,160)

 160      FORMAT (2X, 'None were divisible by 3.')
      ELSE IF (RESULT.EQ.1) THEN
      WRITE (*,170)

 170  FORMAT (2X, 'One was divisible by 3.')

      ELSE
         WRITE (*,180) RESULT

 180  FORMAT(2X, F7.2,'  were divisible by 3.')
      END IF

      STOP

      END


      DOUBLE PRECISION FUNCTION SUM(A,N)
C     .. Scalar Arguments ..
      INTEGER N
C     ..
C     .. Array Arguments ..
      INTEGER A(N)
C     ..
C     .. Local Scalars ..
      INTEGER I
C     ..
      SUM = 0
      DO 100 I = 1, N
 100  SUM = SUM + A(I)
      END

      INTEGER FUNCTION NEVEN(A,N)
C     .. Scalar Arguments ..
      INTEGER N
C     ..
C     .. Array Arguments ..
      INTEGER  A(N)
C     ..
C     .. Local Scalars ..
```

```
      INTEGER I
C     ..
C     .. Intrinsic Functions ..
      INTRINSIC MOD
C     ..
      NEVEN = 0
      DO 100 I = 1,N
  100 IF (MOD(A(I),2).EQ.0) NEVEN = NEVEN + 1
      END

      INTEGER FUNCTION ND3(A,N)
C     .. Scalar Arguments ..
      INTEGER N
C     ..
C     .. Array Arguments ..
      INTEGER A(N)
C     ..
C     .. Local Scalars ..
      INTEGER I
C     ..
C     .. Intrinsic Functions ..
      INTRINSIC MOD
      ND3 = 0
      DO 100 I = 1, N
  100 IF (MOD(A(I),3).EQ.0) ND3 = ND3 + 1
      END
```

APPENDIX E

THE COMMANDS AND OUTPUTS (ISTAL)

1. ISTAL COMMAND

This appendix contains a quick reference guide to the commands available in ISTAL.

The following commands control the general operation of ISTAL:

- DEbug (=YES/NO)
- FOlding (=YES/NO)
- Intrinsics (=YES/NO)
- Verbose (=YES/NO)

The following commands are based on the use of information from ISTYP format symbol tables:

- CAllgraph (= #file name)
- COmmon (= #file name)
- FUllxreference (= #file name)
- SYmbol (= expression)
- TAble (= #file name)
- Warning (= expression)
- Xreference (= #file name)

The following commands are based on the use of information derived either from the analyzer(ISTAN) or the results of an instrumented run:

- ANnotated (= #file name)
- Run time (= #file name)
- SUmmary (= #file name)
- ASsertions (= expression)
- Listing (=no<list>)
- SEgments (= expression)
- STatic (= expression)
- TOtals (= expression)

2. The output of CALLGRAPH

```
The following callgraph shows the routine dependencies
of those routines and entry points detailed within the
specified symbol table files.
Where an entry is followed by a number in brackets, the
number refers to the line on which that entry's
expansion has already been shown.
Question mark, this indicates that the routines symbol
tables was 126 provided.

   1 TEST
   2     MEAN
   3     SQUARE
   4     STAND
   5         MEAN
   6         SQRT (Std. Intrinsic)
   7     SUM
al:
```

3. The output of COMMON command

```
   The following table details the usage of common blocks
   within the specified symbol table files.
   Each common block is given, followed by the name of the
   block data program 3 it appears in (if relevant).
   $common is unnamed common,
   $BLOCKDATA is unnamed block data.

   There are no common blocks used.

   al :
```

4. The output of WARNING command

```
The following table shows warnings drived from the
symbol tables of the specified program units.


Warnings for program unit : STAND
  Untyped Variable : I
```

5. The output of SYMBOL = TEST command

```
The following table shows the symbol usage for the
specified program units.

Symbol table information for program unit : TEST
     Variables :
          A  - REAL (declared as an array)
               Explicitly typed
               In READ input list
               Used as an actual argument
               In an expression
          X  - REAL
               Assigned to on LHS of "="
               In a expression
     Procedures :
          MEAN  - REAL
               Explicitly  typed
               Called as a function
               In an expression
          SQUARE  - Routine
               Called as a subroutine
          STAND  - REAL
               Explicitly typed
               Called as a function
               In an expression
          SUM  - REAL
               Explicitly typed
               Called as a function
               In an expression

al :
```

6. The output of SYMBOL = STAND command

```
The following table shows the symbol usage for the
specified program units.

Symbol table information for program unit :  STAND
     Variables :
          A - REAL (declared as an array)
              Formal parameter
              Explicitly typed
              Used as an actual argument
              In an expression
          I - INTEGER
              In an expression
              Used as a DO-loop index
          M - REAL
              Explicitly typed
              Assigned to on LHS of "="
              In an expression
          N - INTEGER
              Formal parameter
              Explicitly typed
              Used as an actual argument
              In an expression
     Procedures :
          MEAN - REAL
              Explicitly typed
              Called as a function
              In an expression
          SQRT  - Generic
              Standard intrinsic function
              Called as a function
              In an expression

al :
```

7. The output of XREFERENCE command

```
The following sub-section show the routine dependencies
of those routines and entry points detailed within the
specified symbol table files.

TEST
            NOT CALLED

MEAN
            CALLED BY:
                TEST, STAND

SQUARE
            CALLED BY:
                TEST

STAND
            CALLED BY:
                 TEST

SUM
            CALLED BY:
                   TEST

SQRT
            CALLED BY:
                   STAND

al :
```

8. The output of FULLXREFERENCE command

```
The following sub-section show the routine dependencies
of those routines and entry points detailed within the
specified symbol table files.

TEST
            CALLS:
                  MEAN,SQUARE,STAND,SUM
            NOT CALLED

MEAN
            CALLS NOTHING
            CALLED BY:
                  TEST, STAND

SQUARE
            CALLS NOTHING
            CALLED BY:
                  TEST

STAND
            CALLS:
                  MEAN,SQRT
            CALLED BY:
                  TEST

SUM
            CALLS NOTHING
            CALLED BY:
                  TEST

SQRT
            {Standard Intrinsic}
            CALLED BY:
                  STAND

al :
```

APPENDIX F

THE OUTPUTS OF ISTAN

1. Fortran-77 SOURCE PROGRAM

```
      PROGRAM TEST
      REAL A(10), MEAN, SUM, STAND
      PRINT *,'This is a sample test program'
      PRINT *,'Please input 10 real numbers:'
      READ *,A
      X=MEAN(A,10)
      PRINT *,'The mean is :',X
      X=STAND(A,10)
      PRINT *,'The standard deviation is :',X
      CALL SQUARE(A,10)
      PRINT *,'The square of the numbers are :', A
      X=SUM(A,10)
      PRINT *,'The sum of the squares is :',X
      X=STAND(A,10)
      PRINT *,'The standard deviation is :',X
      STOP
      END

      SUBROUTINE SQUARE(A,N)
      REAL A(N)
      INTEGER I,N
      DO 100 I=1,N
      A(I) =A(I) * A(I)
100   CONTINUE
      RETURN
      END

      REAL FUNCTION SUM(A,N)
      REAL A(N)
      INTEGER I,N
      SUM=0
      DO 100 I=1,N
      SUM = SUM + A(I)
100   CONTINUE
      RETURN
      END
      REAL FUNCTION MEAN(A,N)
      REAL A(N)
      MEAN=0
      DO 100 I=1,N
100   MEAN=MEAN/N
      END

      REAL FUNCTION STAND(A,N)
      REAL A(N)
      INTEGER N
      REAL M,MEAN
      M=MEAN(A,N)
      STAND=0
      DO 100 I=1,N
 100  STAND=STAND+(A(I)-M)**2
      STAND=SQRT(STAND/N)
      IF (STAND.EQ.0) RETURN
      END
```

2. The Instrumented program execution result

```
   This is a sample test program
   Please input 10 numbers :
   1 2 3 4 5 6 7 8 9 10
   The mean is : 5.500000
   The standard deviation is : 2.872281
   The square of the numbers are :
   1.00000  4.00000  9.00000  16.00000  25.00000
   36.00000 49.00000 64.00000 81.00000  100.00000
   The sum of the squares is : 385.0000
   The standard deviation is : 32.41990


            SEGMENT EXECUTION FREQUENCIES - CURRENT

          0    1    2    3    4    5    6    7    8    9
   TEST
0x             1
   SQUARE
0x                  1    10   1
   SUM
0x                                 1    10   1
   MEAN
0x                                                  3    30
1x        3
   STAND
1x             2    20   2    0    2
          SEGMENT NOT EXECUTED
14
FORTRAN STOP
```

3. The output of LISTING command

The following listing of the instrumented program has been annotated with the segment execution frequencies and assertion failure counts taken from the file :

```
#SAMPLE.DAT
SEGMENT 1:  1
                      PROGRAM TEST
                      REAL A(10), MEAN, SUM, STAND
                      PRINT *,'This is a sample test program'
                      PRINT *,'Please input 10 real numbers:'
                      READ * ,A
                      X=MEAN(A,10)
                      PRINT *,'The mean is :',X
                      X=STAND(A,10)
                      PRINT *,'The standard deviation is :',X
                      CALL SQUARE(A,10)
                      PRINT *,'The square of the numbers are :
                        A
                      X=SUM(A,10)
                      PRINT *,'The sum of the squares is :',X
                      X=STAND(A,10)
                      PRINT *,'The standard deviation is :',X
                      STOP

                      END

SEGMENT 2:  1
                      SUBROUTINE SQUARE(A,N)
                      REAL A(N)
                      INTEGER I,N

                      DO 100 I = 1,N
SEGMENT 3:  10
                          A(I) = A(I) * A(I)
                100   CONTINUE
SEGMENT 4:  1
                      RETURN

                      END
SEGMENT 5:  1
                      REAL FUNCTION SUM(A,N)
                      REAL A(N)
                      INTEGER I,N

                      SUM = 0
                      DO 100 I = 1,N
SEGMENT 6:  10
                          SUM = SUM + A(I)
                100   CONTINUE
SEGMENT 7:  1
                      RETURN
                      END

SEGMENT 8:  3
                      REAL FUNCTION MEAN(A,N)
                      REAL A(N)

                      MEAN = 0
                      DO 100 I = 1,N
SEGMENT 9:  30
                100   CONTINUE
SEGMENT 10:  3
                      MEAN = MEAN/N
                      END
```

```
SEGMENT 11: 2
                    REAL FUNCTION STAND(A,N)
                    REAL A(N)
                    INTEGER N
                    REAL M,MEAN

                    MEAN = MEAN(A,N)
                    STAND = 0
                    DO 100 I = 1,N
SEGMENT 12 :20
          100       STAND = STAND + (A(I)-M)**2
SEGMENT 13 :2
                    STAND = SQRT(STAND/N)
SEGMENT 14 :0
                    IF (STAND.EQ.0) RETURN
SEGMENT 15 :2
                    END
```

4. The output of SEGMENT = ?*

The following table shows the execution frequencies for the various segments. The first count for each program 3 is also the invocation frequency for that 3.

```
SEGMENT EXECUTION FREQUENCIES

NAME    FIRST  SEG        EXECUTION FREQUENCIES
-----------------------------------------------

TEST      (  1) :     1
SQUARE    (  2) :     1,    10,   1
SUM       (  5) :     1,    10,   1
MEAN      (  8) :     3,    30,   3
STAND     ( 11) :     2,    20,   2,    0,     2
```

5. The output of TOTALS = ?*

The following table gives information derived from the static and dynamic statistics specified.

SUMMARY TOTALS

--PROGRAM UNIT--		------STATEMENTS------			---SEGMENTS---	
NAME	INVOCATION FREQUENCY	TOTAL NUMBER	EXEC-UTABLE	PERCENT EXECUTED	TOTAL NUMBER	PERCENT EXECUTED
TEST	1	17	15	100	1	100
SQUARE	1	8	5	100	3	100
SUM	1	9	6	100	3	100
MEAN	3	7	5	100	3	100
STAND	2	11	7	100	5	80
-TOTAL	8	52	38	100	15	93

6. The output of STATIC = TEST

This table contains a count of the statements in the specified program 3, split by statement type.

STATIC SUMMARY FOR PROGRAM UNIT : TEST

```
ASSERTIONS :   0
COMMENTS   :   1
ERRORS     :   0
TOKENS     : 114
STATEMENTS :  17
```

ASSIGN	0	GO TO	0
BACKSPACE	0	--(ASSIGNED)	0
BLOCK DATA	0	--(COMPUTED)	0
CALL	1	--(UNCONDITIONAL)	0
CHARACTER	0	IF	0
CLOSE	0	--(ARITHMETIC)	0
COMMON	0	--(BLOCK)	0
COMPLEX	0	LOGICAL	0
CONTINUE	0	IMPLICIT	0
DATA	0	INQUIRE	0
DIMENSION	0	INTEGER	0
DOUBLE PRECISION	0	INTRINSIC	0
DO	0	LOGICAL	0
ELSE IF	0	OPEN	0
ELSE	0	PARAMETER	0
ENDFILE	0	PAUSE	0
END IF	0	PRINT	7
END	1	PROGRAM	1
ENTRY	0	READ	1
EQUIVALENCE	0	REAL	1
EXTERNAL	0	RETURN	0
FORMAT	0	REWIND	0
FUNCTION	0	SAVE	0
--CHARACTER	0	STOP	1
--COMPLEX	0	SUBROUTINE	0
--DOUBLE PRECISION	0	WRITE	0
--INTEGER	0	(ASSIGNMENT STATEMENTS)	4
--LOGICAL	0	(STATEMENT FUNCTIONS)	0
--REAL	0	(UNRECOGNIZED STATEMENTS)	0
--UNTYPED	0	-	0

7. The output of DYNAMIC = TEST

This table contains a count of the statements actually executed in the specified program 3, split by statement type.

DYNAMIC SUMMARY FOR PROGRAM UNIT : TEST

ASSIGN	0	IF	2
BACKSPACE	0	--(ARITHMETIC)	0
CALL	1	--(BLOCK)	0
CLOSE	0	--(LOGICAL)	2
CONTINUE	0	INQUIRE	0
DO	0	OPEN	0
ELSE IF	0	PAUSE	0
ELSE	0	PRINT	0
ENDFILE	0	READ	0
END IF	0	RETURN	0
END	1	REWIND	0
GO TO	0	STOP	0
--(ASSIGNED)	0	WRITE	0
--(COMPUTED)	0	(ASSIGNMENT STATEMENTS)	26
--(UNCONDITIONAL)	0	(UNRECOGNIZED STATEMENTS)	0

LIST OF REFERENCES

1. Cowell, Wayne R. and Miller, Webb C., The TOOLPACK Prospectus, Argonne National Laboratory, 1979.

2. Petersen, Perry, Project Control System, Datamation 25 June 1979.

3. Boehm, Barry W., Software Engineering Economics, Prentice-Hall, Inc., 1981.

4. Ford,B., Rault,J.C. and Thomasset,F., Tools,Methods and Languages for Scientific and Engineering Computation, Elsevier Science Publishers B.V., 1984.

5. Boyle, J.M. and Dritz, K.W., An Automated Programming System to facilitate the Development of Quality in Mathematical Software, Information Processing 74, North-Holland publishing Company, 1974.

6. Fosdick, L. D. and Osterweil, L. J., Data flow analysis in software reliability, pp. 305-330, ACM Computing Surveys 8, (3), 1976.

7. Osterweil, Leon J., Hague, Stephen, and Miller, Webb., TOOLPACK Architectural Design: The Users' Perspective, Argonne National Laboratory, 1982.

8. Wenger,Peter(editor), Research Directions in Software Technology, MIT Press, Cambridge, MA, 1979.

9. Kernighan, Brian W. and Plauger, P.J., Software Tools, Addison-Wesley Publishing Company, 1976.

10. Meyer, B. Priciples of Package Design, Communication ACM, VOL. 25, no7, July 1982.

11. Vick, C.R. and Ramamoorthy, C.V., Handbook of Software Engineering, Van Nostrand Reinhold Company, 1984.

12. Senn, James A., Analysis and Design of Information Systems, McGrow_Hill Book Company, 1984.

13. R.M.J. Iles, TIEECODE Installer's Guide, NAG Technical Memorandum : NAG/T6-TIG, 1985.

14. Cowell, W.R., Hague, S.J. and Iles, R.M.J., TOOLPACK/1 Introductory Guide, 1985.

15. ISTLX-Fortran 77 Scanner Users' Guide, NAG Technical Memorandum: NAG/T24-TLX, 1985.

16. Moore, John B. and Makela, Leo J., Structured Fortran with WATFIV, Reston Publishing Company, Inc., 1981.

17. DeMarco, Tom, Structured Analysis and System Specification, Yourdon Inc, 1978.

18. ISTYP-Fortran 77 Parser Users' Guide, NAG Technical Memorandum: NAG/T38-TYP, 1985.

19. Iles, R.M.J. and Hague, S.J., TOOLPACK : The first public release, Numerical Algorithms Group, Oxford, UK, 1985.

20. ISTPL/ISTPO Users' Guide, NAG Technical Memorandum: NAG/T34-TPL, 1985.

21. ISTDS-Declaration Standardiser Users' Guide, NAG Technical Memorandum: NAG/T36-TDS, 1985.

22. VAX Fortran Users' Guide, Digital Equipment Corporation, 1984.

23. ISTPT-Precision Transformer Users' Guide, NAG Technical Memorandum: NAG/T19-TPT, 1985.

24. ISTAL-Documentation Generation Aid Users' Guide, NAG Technical Memorandum: NAG/T41-TAL, 1985.

25. ISTAN-Execution Analyser Users' Guide, NAG Technical Memorandum: NAG/T42-TAN, 1985.

26. ISTCE-TOOLPACK Command Executor Users' Guide, NAG Technical Memorandum: NAG/T3-TCE, 1985.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Montery, California 93943-5000	2
3.	Computer Technology Programs,Code 37 Naval Postgraduate School Monterey, California 93943-5000	1
4.	Prof. Gordon H. Bradley, Code 52BZ Department of Computer Science Naval Postgraduate School Monterey, California 93943	10
5.	LCDR. Paul Callahan, Code 52CS Department of Comptuer Science Naval Postgraduate School Monterey, California 93943-5000	1
6.	Naval Academy Library Jinhae City, Gyungnam 602-00 Republic of Korea	2
7.	LT. Kim, Jung Sik 25/4 300-51, SADANG 4 DONG, DONG-JAK KU, SEOUL, Republic of Korea	6
8.	LT. Hur, Seong Pil SMC 2246 NPGS Monterey CA 93943-5016	1

END

12-86

DTIC